FACULDADE DE TECNOLOGIA DE AMERICANA

Curso Superior de Tecnologia em Jogos Digitais

GIOVANNI PREGNOLATO ROSIM

DESENVOLVIMENTO DE UM JOGO ADAPTATIVO UTILIZANDO TÉCNICAS DE ALGORITMOS GENÉTICOS

Americana, SP

2014

FACULDADE DE TECNOLOGIA DE AMERICANA
Curso Superior de Tecnologia em Jogos Digitais

GIOVANNI PREGNOLATO ROSIM
giovanni_pr@hotmail.com

DESENVOLVIMENTO DE UM JOGO ADAPTATIVO UTILIZANDO TÉCNICAS DE ALGORITMOS GENÉTICOS

Trabalho Monográfico, desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Jogos Digitais da Fatec-Americana, sob orientação do Prof. Me. Vitor Brandi Junior.

Área: Agentes Inteligentes, Algoritmo Genético, Inteligência Artificial.

Americana, SP
2014

BIBLIOTECA Fatec Americana

Formulário de Solicitação de Ficha Catalográfica (FC) – 2013
Dados Internacionais de Catalogação-na-fonte

Dados pessoais

Nome completo: Giovanni Pregnolato Rosim	
e-mail pessoal: giovanni_pr@hotmail.com	Fone: (11) 99559-3220
Curso: Jogos Digitais	

Dados da publicação

Autor(es): Giovanni Pregnolato Rosim	
Título e subtítulo: Desenvolvimento de um jogo adaptativo utilizando técnicas de Algoritmo Genético	
Orientador **(nome completo e titulação)**: Vitor Brandi Junior (Mestre)	
Área temática: Ciência da Computação	
N°.de folhas: 69	Avaliação/Nota: 9.5

Resumo e Palavras-chave

A cada ano que passa os jogos digitais vão ficando cada vez mais complexos, assim como seus jogadores que ficam cada vez mais exigentes, esperando a cada jogo uma melhor interação entre eles e a máquina. Para que isso ocorra, é necessário ter uma inteligência artificial muito bem implementada, usando técnicas que mais se adequem ao jogo proposto. A fim de compreender e exemplificar algumas dessas técnicas, este trabalho tem como objetivo principal mostrar o desenvolvimento de um simulador de sobrevivência, utilizando inteligência artificial, com a implementação de agentes inteligentes e o uso do algoritmo genético, para fazer a escolha e cruzamento dos melhores seres colocados até aquele momento no ambiente, para saber se o algoritmo citado, sempre chegou a um resultado satisfatório, independente do tempo de simulação. Neste trabalho serão explicadas as técnicas utilizadas, as ferramentas gráficas e de programação para realizá-lo, assim como será feita uma análise de alguns resultados obtidos através do jogo. A partir disso foi possível concluir que o algoritmo genético implementado funcionou bem, sem levar em consideração o tempo decorrido do jogo.

Palavras Chave: Agentes Inteligentes; Algoritmo Genético; Desenvolvimento de Jogos Digitais.

FACULDADE DE TECNOLOGIA DE AMERICANA
Curso Superior de Tecnologia em Jogos Digitais

GIOVANNI PREGNOLATO ROSIM
giovanni_pr@hotmail.com

DESENVOLVIMENTO DE UM JOGO ADAPTATIVO UTILIZANDO TÉCNICAS DE ALGORITMOS GENÉTICOS

Trabalho Monográfico, desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Jogos Digitais da Fatec-Americana, sob orientação do Prof. Me. Vitor Brandi Junior.

Área: Agentes Inteligentes, Algoritmo Genético, Inteligência Artificial.

Americana, SP
2014

Intruções

- ✓ Preencher devidamente e enviar para biblioteca@fatec.edu.br ;
- ✓ O procedimento de FC é realizado totalmente **via e-mail;**
- ✓ Em caso de mais de uma autoria, favor indicar todos em “Dados da publicação”;
- ✓ Copiar e colar o resumo com palavras-chave de sua monografia na Caixa de texto do formulário;
- ✓ Copiar e colar a página de rosto de sua monografia na Caixa de texto do formulário;
- ✓ O período para a solicitação de FCs é o **mês de julho, de seu 1º dia útil até seu 15º dia útil para os TCCs de 1º semestre;**
- ✓ O período para a solicitação de FCs é o **mês de dezembro, de seu 1º dia útil até seu 15º dia útil para os TCCs de 2º semestre;**
- ✓ Todas as FCs solicitadas serão entregues prontas até o último dia útil do mês do semestre correspondente; e,
- ✓ Não será realizada FC fora deste procedimento.

Obrigada

A BIBLIOTECA

FICHA CATALOGRÁFICA – Biblioteca Fatec Americana - CEETEPS
Dados Internacionais de Catalogação-na-fonte

Rosim, Giovanni Pregnolato
R731d Desenvolvimento de um jogo adaptativo utilizando técnicas de algoritmos genéticos. / Giovanni Pregnolato Rosim. – Americana: 2014.
67f.

Monografia (Graduação de Tecnologia em Gestão empresarial). - - Faculdade de Tecnologia de Americana – Centro Estadual de Educação Tecnológica Paula Souza.
Orientador: Prof. Me. Vitor Brandi Junior

1.Jogos eletrônicos 2. Algoritmos 3. Inteligência artificial I. Brandi Junior, Vitor II. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana.

CDU: 681.6
510.5

GIOVANNI PREGNOLATO ROSIM
giovanni_pr@hotmail.com

DESENVOLVIMENTO DE UM JOGO ADAPTATIVO UTILIZANDO TÉCNICAS DE ALGORITMOS GENÉTICOS

Trabalho de conclusão de curso apresentado à Faculdade de Tecnologia de Americana como parte dos requisitos para obtenção do título de Tecnólogo em Jogos Digitais.
Área de concentração: Inteligência Artificial

Americana, 27 de JUNHO de 2014.

Banca Examinadora:

_____
Vitor Brandi Junior
Mestre
FATEC-Americana

_____
Francesco Artur Perrotti
Mestre
FATEC-Americana

_____
Luciene Maria Garbuio Castello Branco
Mestre
FATEC-Americana

AGRADECIMENTOS

Primeiramente, agradeço à minha família que sempre me apoiou e me deu força para chegar a essa etapa do curso.

Agradeço meu orientador Vitor Brandi Junior pela dedicação e grande ajuda na parte escrita do presente trabalho, sempre fazendo críticas construtivas.

Aos meus colegas de classe, que enfrentaram comigo todas as etapas do curso, trocando conhecimentos e experiências.

Aos meus companheiros das repúblicas por onde morei durante o período do curso, pois cresci muito como pessoa aprendendo a respeitar os diferentes modos de pensar e agir de cada um.

“Que os vossos esforços desafiem as impossibilidades, lembrai-vos de que as grandes coisas do homem foram conquistadas do que parecia impossível.” (Charles Chaplin).

RESUMO

A cada ano que passa os jogos digitais vão ficando cada vez mais complexos, assim como seus jogadores que ficam cada vez mais exigentes, esperando a cada jogo uma melhor interação entre eles e a máquina. Para que isso ocorra, é necessário ter uma inteligência artificial muito bem implementada, usando técnicas que mais se adequem ao jogo proposto. A fim de compreender e exemplificar algumas dessas técnicas, este trabalho tem como objetivo principal mostrar o desenvolvimento de um simulador de sobrevivência, utilizando inteligência artificial, com a implementação de agentes inteligentes e o uso do algoritmo genético, para fazer a escolha e cruzamento dos melhores seres colocados até aquele momento no ambiente, para saber se o algoritmo citado, sempre chegou a um resultado satisfatório, independente do tempo de simulação. Neste trabalho serão explicadas as técnicas utilizadas, as ferramentas gráficas e de programação para realizá-lo, assim como será feita uma análise de alguns resultados obtidos através do jogo. A partir disso foi possível concluir que o algoritmo genético implementado funcionou bem, sem levar em consideração o tempo decorrido do jogo.

Palavras Chave: Agentes Inteligentes; Algoritmo Genético; Desenvolvimento de Jogos Digitais.

ABSTRACT

Every year digital games are becoming increasingly complex, even as their players become increasingly demanding, expecting each game better interaction between them and the machine. For this, it is necessary to have a well implemented artificial intelligence, using techniques that best suit to the proposed game. In order to understand and exemplify some of these techniques, this work has as main goal to show the development of a survival simulator, using artificial intelligence, with the implementation of intelligent agents and the use of genetic algorithm to make the choice and cross the best being placed up to that point in the environment, to make sure if such the cited algorithm, has always achieved a satisfactory result, regardless the simulation time. The techniques, the graphical tools and programming tools for this work will be presented, as well as an analysis of some obtained results through the game. From this it was possible to conclude that genetic algorithm implemented worked well, without considering the elapsed time of the game.

Keywords: Intelligent Agents; Genetic Algorithm; Digital Game Development.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

AG	Algoritmo Genético
API	*Application Programming Interface*
FPS	*First Person Shooter*
IA	Inteligência Artificial
IDE	*Integrated Development Environment*
NPC	*Non-Player Character*
PX	*Pixel*
QI	Quociente de Inteligência
UML	*Unified Modeling Language*

SUMÁRIO

1 Introdução

Neste capítulo será apresentada a introdução do trabalho, que é dividido em considerações iniciais, objetivos, justificativas e organização do trabalho.

1.1 Considerações Iniciais

O presente trabalho diz respeito a implementação de um jogo de sobrevivência em uma ilha, usando técnicas de agentes inteligentes para a personagem principal e algoritmo genético para obter os melhores seres e os cruzar, com o intuito de melhorar as seguintes gerações.

O objetivo é verificar se o algoritmo genético foi realmente eficiente e chegou a um resultado satisfatório independente do tempo de simulação, expondo alguns resultados obtidos.

No decorrer do trabalho é mostrado o desenvolvimento do mesmo, com uma breve explicação sobre orientação a objeto em C++, assim como explicar a escolha do OpenGL para gerenciar a parte gráfica no simulador.

1.2 Objetivos

Serão listados a seguir, os objetivos que deseja-se alcançar ao final deste trabalho:

- Estudar os assuntos necessários para embasar a pesquisa: Algoritmos Genéticos, Agentes Inteligentes, Técnicas de Programação em C++ e OpenGL.
- Implementar o simulador utilizando técnicas de Inteligência Artificial, em específico, Algoritmos Genéticos e Agentes Inteligentes.
- Recolher as informações de pontuação de cada agente, nas simulações realizadas.
- Analisar e expor os resultados mais significativos que foram obtidos.
- Verificar se o Algoritmo Genético sempre obteve um resultado

satisfatório.

O tipo de pesquisa desenvolvida para alcançar esses objetivos, foi de cunho experimental, com busca em fontes de informação bibliográfica, que permitiram o desenvolvimento do estudo de caso, fornecendo dados quantitativos cuja a análise permitiu comprovar as premissas do trabalho.

1.3 Justificativa

A IA sempre foi muito importante para os jogos, pois é através dela que os personagens controlados por computadores, ou NPCs, ganham vida e se tornam mais realistas na maneira de agir ou pensar, com a intenção de trazer mais dificuldade ao jogo, ou mesmo ajudar o jogador a cumprir seu objetivo.

Exemplos são os jogos das séries *Resident Evil Outbreak* (*Capcom*) e *Left 4 Dead* (*Valve*), que se for jogado de maneira *single player*, o computador irá controlar os aliados da personagem principal.

Porém, no caso do *Resident Evil Outbreak*, os NPCs não apresentam uma autonomia, pois logo nos primeiros inimigos que aparecem no jogo, se o jogador não ajudá-los, eles acabam morrendo, não trazendo nenhuma vantagem para quem está jogando.

Já em *Left 4 Dead*, os personagens controlados pelo computador conseguem resistir mais aos inimigos, porém não apresentam estratégias e, quando o jogador precisa de ajuda dos mesmos, muitas vezes acabam deixando-o morrer.

Com esses dois exemplos em mente, uma melhor IA dos NPCs pode ser a diferença entre jogar apenas dez minutos ou conseguir chegar até o final. Para esses casos, poderiam ser aplicados algumas formas de implementação da IA, como por exemplo o algoritmo genético que faz a escolha dos melhores NPCs.

Depois de um certo tempo, cruzariam suas habilidades, atributos ou conhecimentos para criar um descendente melhor e mais apto a ajudar o jogador durante a partida, se adequando ao modo do mesmo jogar.

No caso do aprendizado durante a partida, usaria-se a técnica de implementação de um Agente Inteligente para, durante a própria partida, distinguir as coisas que fez errado e que tiveram consequências negativas e depois de um tempo não voltar a repetir. Ou quando fizer algo certo e aprender que deve-se continuar fazendo isso, pois ajudou o jogador durante o jogo.

Os jogadores estão cada vez mais exigentes, querendo sentir, por exemplo, que quando têm uma equipe de NPCs ao seu dispor, onde ela corresponda da melhor forma possível, como ao enfrentam um inimigo, espera-se que os mesmos lhe tragam alguma dificuldade, usando diferentes estratégias para derrotar quem está jogando.

Jogos como Pong, Mario e Sonic, por mais conceituados que sejam não têm uma inteligência artificial refinada. As ações que ocorrem dentro destes jogos, vindas dos NPCs são sempre parecidas, como por exemplo seguir o jogador ou a bola (no caso do Pong) e andar de um lado para outro, mas isso não é tão atraente e hoje se faz necessário que as personagens controladas pelo computador terem uma IA bem implementada.

1.4 Organização do Trabalho

Este trabalho é dividido em cinco capítulos, o primeiro apresenta a introdução, dando uma breve explicação sobre inteligência artificial, com sub-tópicos descrevendo objetivos, considerações iniciais e a justificativa pela escolha do tema.

No segundo capítulo, tem-se a revisão bibliográfica, onde se explica de forma abrangente a IA, para situar o leitor sobre o tema, mostrando de forma mais detalhada as técnicas utilizadas no presente trabalho, Algoritmos Genéticos e Agentes Inteligentes, apresentando referências bibliográficas sobre as mesmas. Por fim, uma breve explicação sobre programação orientada a objeto, usando exemplos do próprio código do simulador, através da linguagem de programação C++, e o porquê a API gráfica OpenGL foi escolhida para o desenvolvimento do presente trabalho.

No terceiro capítulo, apresenta-se a proposta de trabalho, contando com uma breve explicação do que será realizado no simulador.

No quarto capítulo, desenvolvimento do jogo, mostra-se a implementação do jogo de maneira detalhada, seu objetivo, funcionamento, os objetos encontrados no ambiente, como foram utilizadas as técnicas de algoritmo genético e agente inteligente, como foi feita a pontuação e por fim mostrar resultados obtidos durante a execução do simulador.

Finalizando, o último capítulo com a conclusão do trabalho, trazendo algumas propostas para trabalhos futuros.

2 Revisão Bibliográfica

Para o desenvolvimento do presente trabalho foram usadas duas técnicas para a implementação da IA na personagem, Algoritmo Genético e Agente Inteligente, onde as mesmas foram implantadas no simulador através da linguagem de programação C++, junto com o OpenGL, para ter uma melhor ilustração do que esta acontecendo no ambiente.

2.1 Introdução à Inteligência Artificial

A IA, segundo Priberam (2013), é a área da Informática que estuda o desenvolvimento dos sistemas de computadores, usando como base o conhecimento da inteligência dos seres humanos, ou seja, a programação de uma inteligência artificial busca ao máximo a semelhança com a inteligência dos seres humanos, com a forma de agir ou com a forma de pensar, para se adaptar melhor à uma situação.

Mesmo assim, sobre o QI da melhor IA, usando o sistema *ConceptNet* 4[1], considerada como a inteligência artificial mais apta para os testes, foram feitos testes de vocabulário e de reconhecimento de similaridades. O desempenho do sistema foi muito bom, entretanto, nos testes de compreensão (perguntas de porque), seu desempenho foi dramaticamente pior. Como resultado e conclusão da experiência, foi avaliado que o QI do *ConceptNet* 4 é igual ao de uma criança de 4 anos de idade (Sloan, 2013).

Para entender melhor a Inteligência Artificial foram feitas algumas definições, organizadas em quatro categorias (Russel e Norvig, 2004, p. 5):

- Sistemas que pensam como seres humanos: “o novo e interessante esforço para fazer os computadores pensarem... máquinas com mentes, no sentido total e literal” (Russel (2004) apud Haugeland

[1] Sistema de IA desenvolvido na M.I.T., atráves de colcaborações de verba da Wechsler Preschool e Primary Scale of Intelligence Test

(1985)), neste caso é necessário um estudo sobre o funcionamento da mente do ser humano, para cada caso que for usado este tipo de sistema.

- Sistemas que atuam como seres humanos: “a arte de criar máquinas que executam funções que exigem inteligência quando executadas por pessoas” (Russel (2004) apud Kurzweil (1990)), ou seja, o mesmo deve conseguir interpretar a linguagem natural, para responder as mensagens dadas pelo interrogador, usar o conhecimento que já adquiriu, aprender e tirar suas conclusões.

- Sistemas que pensam racionalmente: “o estudo das computações que tornam possível perceber, raciocinar e agir” (Russel (2004) apud Winston (1992)), de modo que o sistema tenha suas conclusões corretas sobre os fatos.

- Sistemas que atuam racionalmente: “a Inteligência Computacional é o estudo do projeto de agentes inteligentes” (Russel (2004) apud Poole et al. (1998)), que em resumo é, o sistema ter autonomia, sendo capaz de perceber o ambiente e se adaptar ao mesmo.

Ou seja, para começar a desenvolver um programa que usa IA, deve-se primeiro saber qual será a categoria do sistema utilizado. Para o simulador do presente trabalho, foram usados os agentes inteligentes, que são sistemas que atuam racionalmente.

Porém, mesmo sabendo disso, é necessário conhecer algumas estratégias de busca com informação e exploração, segundo Russel (2004), utilizando o conhecimento específico do problema. Pode-se encontrar soluções de modo mais eficiente, algumas delas são:

- Busca A*: é o método de busca em grafos, que avalia a quantidade de nós combinados, seu custo para chegar a cada nó e o custo para

chegar até o objetivo, de modo a escolher o caminho com o custo mais baixo.

- Busca de Subida de Encosta: este algoritmo é feito apenas por um laço em uma árvore, que tem seu fim quando é alcançado o “pico”, no qual nenhum vizinho tem um melhor resultado para o problema.

- Busca em Feixe Local: é um algoritmo que tem o controle sobre uma quantidade indeterminada de estados; que em seu começo são gerados aleatoriamente. Em cada passo são gerados os sucessores de cada um dos estados, se algum deles for o objetivo, o algoritmo irá parar. Como exemplo deste método de busca, tem-se o Algoritmo Genético.

Nos jogos, a inteligência artificial obteve uma evolução desde seu início, usando de exemplo os jogos de futebol, como o Internacional *Super Star Soccer* (Konami), que foi um dos primeiros jogos do gênero a fazer sucesso.

Por mais refinada que fosse a IA para época, as jogadas e as estratégias executadas pelas máquinas eram sempre parecidas. Nos jogos mais modernos, como *FIFA* (EA Sports) e *Pro Evolution Soccer* (Konami), as estratégias e jogadas se adaptam ao momento do jogo e a formação do adversário, proporcionando, uma grande variedade recursos.

Tendo isso como base, a inteligência artificial tem ainda uma grande área nos jogos para ser explorada, como nos *FPS*, onde os inimigos não são capazes de elaborar estratégias sofisticadas, ou entender padrões do modo de jogar do jogador para criar melhores táticas e enfrentá-lo da melhor maneira.

2.1.1 Algoritmo Genético

O Algoritmo Genético é um método de busca e otimização, segundo Belisario (2010). O nome genético não se deu ao acaso, ele é inspirado no modelo de evolução ou seleção natural de Charles Darwin.

Usando um breve exemplo para explicar a Seleção Natural, através da pergunta do porque a girafa têm o pescoço comprido:

Segundo Paul (2003) apud Darwin (1859), “A Origem das Espécies”, não foi pelo motivo das girafas ficarem sempre esticando seu pescoço para buscar as folhas mais altas de uma árvore, mas sim, por causa de umas serem maiores do que as outras, o que as tornavam as melhores para aquele ambiente, fazendo com que aos poucos, as girafas menores fossem morrendo ou migrando para outros lugares.

Resumindo, seleção natural é a escolha do próprio ambiente, onde os seres mais adaptados sobrevivem e podem se reproduzir, gerando descendentes que também estão aptos para sobreviverem no local.

Como é afirmado por Russel (2004), os AGs são um conjunto de indeterminados estados ou indivíduos que são gerados aleatoriamente, sendo chamado de população.

Para produzir a próxima geração de seres, cada indivíduo passa por uma função de avaliação, ou como é apresentado na terminologia do AG, função fitness, onde são retornados os maiores valores dos melhores estados. Desta forma, quanto maior a pontuação fitness, maior as chances de ser escolhido para gerar os descendentes da próxima geração.

Belisario (2010), pondera que após a escolha dos estados, acontece a reprodução, que é dividida em três etapas:

- Acasalamento: é a escolha dos dois seres mais aptos a ter descendentes.

- Recombinação: também conhecida como *crossover*, é o modo de reprodução sexuada, onde os descendentes que são gerados, recebem no seu genótipo uma parte do código genético da mãe e do pai, fazendo trocas das características dos seres mais aptos, para a cada geração criar seres mais adaptados.

- Mutação: em cada recombinação a chance de acontecer a mutação é muito pequena, mas com isso, garante-se uma variedade da espécie.

Sobre *crossover*, “a principal vantagem dos algoritmos genéticos, se existir, vem da operação de *crossover*, para combinar grandes blocos de genes que evoluem de forma independente para executar funções úteis, elevando assim o nível de afunilamento em que a busca opera” (Belisario, 2010), ou seja, com base em Russel (2004), como no início a população ainda é bastante diversa, por causa dos primeiros seres terem seus atributos sorteados, usando o *crossover* pode se agilizar o processo para chegar ao resultado desejado, contudo, a cada geração que passa a evolução se torna cada vez menor, pois os estados já estão muito parecidos.

A imagem a seguir mostra um exemplo de *crossover* de um ponto, onde um bloco de genes do pai, se junta com outro bloco de genes da mãe:

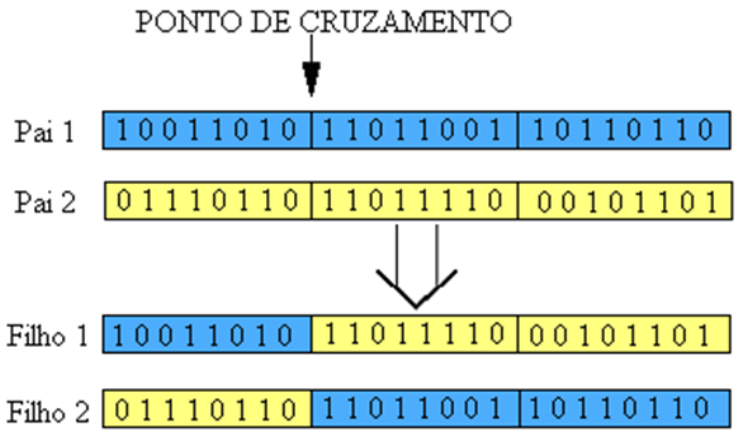
Figura 1 - Crossover de um ponto

Fonte: Seer – Repositório Digital da UFU[1].

No caso de um exemplo de *crossover* de um ponto, representado numericamente, o mesmo ficaria assim, conforme a figura:

[1] Disponível em: http://www.seer.ufu.br/index.php/horizontecientifico/article/viewFile/4050/3015; Acesso em jun. 2014.

Figura 2 – *Crossover* Binário



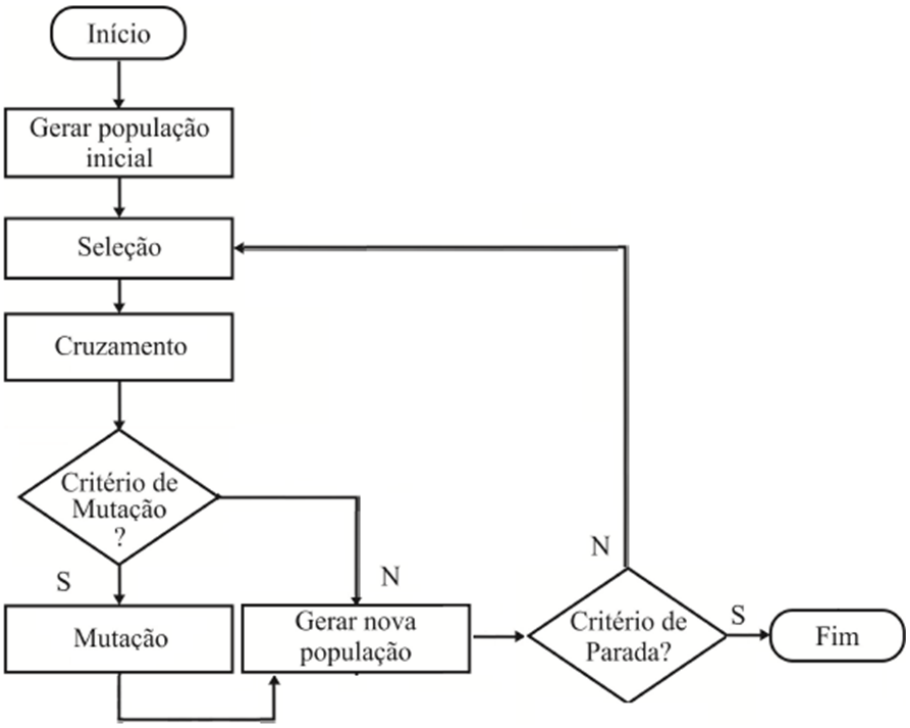
Fonte: http://rived.mec.gov.br/atividades/biologia/externos/AGPM/AGPMaula1.htm

Explicando o que foi mostrado nas imagens anteriores, o Pai 1, representado pela fita azul, e o Pai 2, fita amarela, tem seus genes de maneira binária, zero e um, com o crossover, um bloco da fita de cada cada pai fica com um dos seus dois filhos, o tamanho deste bloco é dado pelo ponto de cruzamento.

No Filho 1, seus genes são constituídos pelo primeiro bloco de genes do Pai 1 e o segundo e terceiro blocos do Pai 2. Já o Filho 2, fica com o que não foi utilizado pelo Filho 1, ou seja, seus genes vem do primeiro bloco do Pai 2 e do segundo e terceiro bloco do Pai 1.

Para explicar o funcionamento do algoritmo genético de uma forma mais simplificada, mostra-se o esquema em um fluxograma na figura a seguir:

Figura 3 – Fluxograma do Algoritmo Genético



Fonte: Elaborada pelo autor.

Segundo Russel (2004), um pseudocódigo de um algoritmo genético poderia ser representado da seguinte maneira, seguindo a mesma lógica do fluxograma:

Figura 4 – Pseudocódigo do AG

```
função ALGORITMO-GENÉTICO(população, FN-FITNESS) retorna um indivíduo
      entradas: população, um conjunto de indivíduos
            FN-FITNESS, uma função que mede a adaptação de um indivíduo

repita
   nova_população <- conjunto vazio
   para i <- 1 até TAMANHO(população) faça
      x <- SELEÇÃO-ALEATÓRIA(população, FN-FITNESS)
      y <- SELEÇÃO-ALEATÓRIA(população, FN-FITNESS)
      filho <- CRUZAMENTO(x, y)
      se(pequena probabililade aleatória) então filho <- MUTAÇÃO(filho)
   adicionar filho a nova_população
   população <- nova_população
```

```
até algum indivíduo estar adaptado o suficiente ou até ter decorrido tempo
suficiente
retornar o melhor indivíduo em população, de acordo com FN-FITNESS
```

```
função CRUZAMENTO(x, y) retorna um indivíduo
      entradas: x, y, indivíduos pais

      n <- COMPRIMENTOS(x)
      c <- número aleatório de 1 a n
retornar CONCATENA(SUBCADEIA(x, 1, c), SUBCADEIA(y, c+1, n))
```

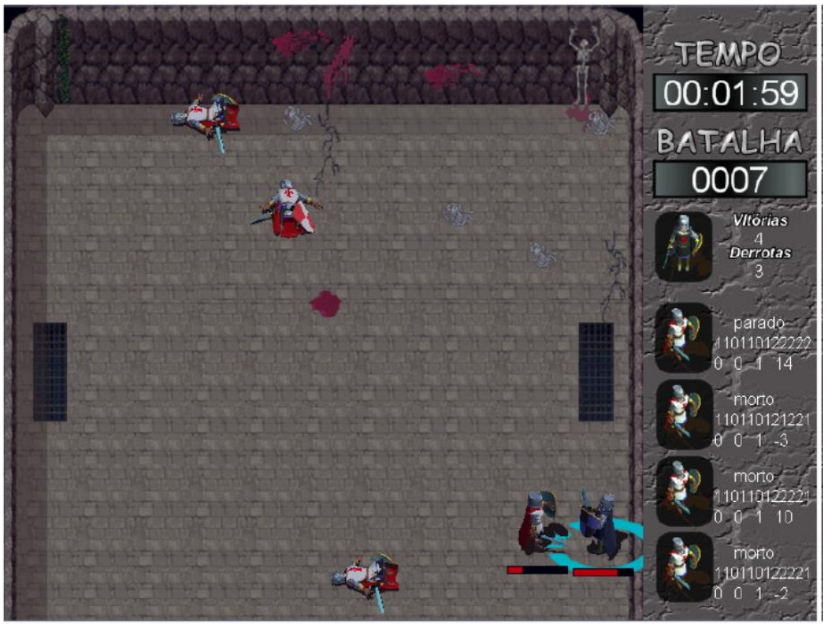
Fonte: RUSSEL, 2004.

A função “ALGORITMO-GENÉTICO”, representa o fluxograma, com a função “CRUZAMENTO” detalhando melhor como ocorre a mesma. Neste caso, o `n` é a quantidade total de genes da cadeia, `c` é o ponto de cruzamento entre os blocos, retornando como resultado da função, a junção entre a primeira subcadeia, que usa os genes do indivíduo `x`, começando do primeiro, até o `c`, e do segundo, o `y`, usando os genes a partir do próximo valor de `c`, `c+1`, como mostrado no código, até o último gene, no caso, o `n`.

2.1.1.1 Exemplo de Aplicação da Técnica

Como exemplo de desenvolvimento de um jogo usando AG, Andrade (2009), em o “O Último Sobrevivente”, tem como foco a evolução dos NPCs através do algoritmo evolutivo ou genético, que após cada batalha, escolhem os dois personagens controlados pelo computador que melhor se adaptaram a batalha e obtiveram o resultado mais satisfatório, e os cruzam com o intuito de adaptar a dificuldade dos mesmos às habilidades e táticas do jogador, para ficar um jogo sempre equilibrado.

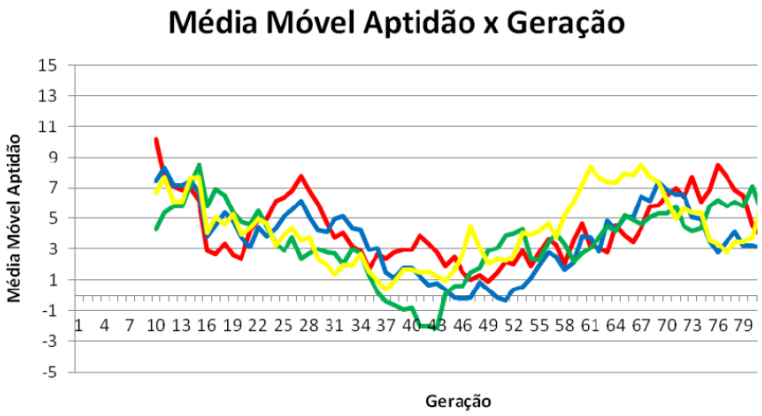
Figura 5 – “O Último Sobrevivente”



Fonte: ANDRADE, 2009.

Alguns resultados foram coletados, e como primeiro teste, após oitenta batalhas, fez-se o gráfico:

Figura 6 – Gráfico do primeiro teste do “O Último Sobrevivente”

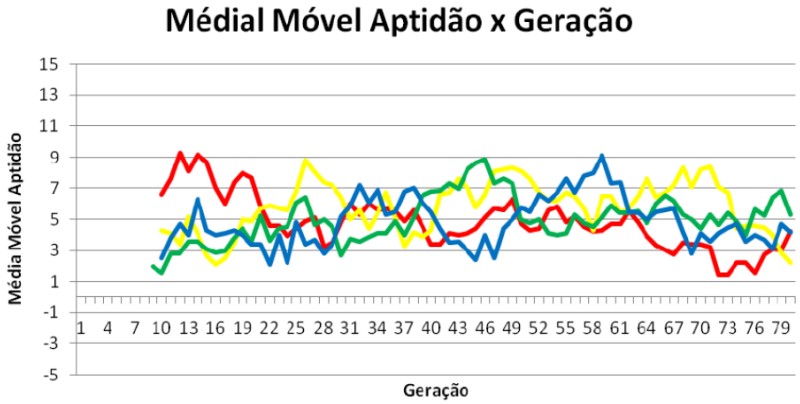


Fonte: ANDRADE, 2009

As linhas de diferentes cores representam cada um dos quatro inimigos e percebe-se que no começo o jogador sente uma certa dificuldade, pois ainda não identificou a estratégia dos inimigos, porém com o tempo, o resultado dos mesmos é pior pelo fato de o jogador ter entendido suas estratégias. Mas, aproximadamente da geração cinquenta e cinco em diante, a pontuação do computador aumenta, pois eles mudaram sua estratégia inicial. Como resultado final, o jogador obteve trinta derrotas e cinquenta vitórias, que foram consecutivas.

Um segundo teste foi realizado, assim como segue no gráfico:

Figura 7 – Gráfico do segundo teste do “O Último Sobrevivente”



Fonte: ANDRADE, 2009

Neste teste, os NPCs aprendiam as estratégias do jogador mais rápido do que no primeiro teste; com isso o jogo ficou mais equilibrado, tanto que o seu resultado final foi de cinquenta e quatro derrotas e vinte seis vitórias, não consecutivas, distribuídas de maneira uniforme.

2.1.2 Agentes Inteligentes

Como o assunto Agentes Inteligentes é muito amplo, são apenas tratados os métodos e conceitos utilizados no simulador que foi desenvolvido ao final deste Trabalho de Conclusão de Curso.

Primeiramente, é importante saber o significado das palavras agente e inteligência, que melhor se adequam ao contexto:

- Agente: segundo Ferreira (1975), agente é aquele que trata de negócios por sua própria conta, ou ainda, segundo Braga (2001), “um agente é algo ou alguém que age em seu benefício”.

- Inteligência: “faculdade de aprender, apreender ou compreender; percepção, apreensão, intelecto, intelectualidade” (Ferreira, 1975), ou seja, é uma pessoa que tem a capacidade de aprender e usar seus conhecimentos para tomar boas decisões, para alcançar o seu objetivo.

Portanto, Agentes Inteligentes são seres que usam sua percepção para registrar as entradas perceptivas a qualquer momento, com isso geram uma sequência de percepções, uma lista do que o agente já registrou. Com esses registros o comportamento do agente, embasado em sua percepção do ambiente, é descrito de maneira abstrata pela função de agente, `f: P* -> A`, mapeando a sequência de percepções (`P*`) para ser tomada uma ação (`A`), conforme Russel (2004).

Para saber se o que agente está fazendo é certo ou errado, faz-se necessário implementar algum método para pontuar tanto as tomadas de decisão boas quanto as ruins para que, com isso, a personagem saiba o que deve continuar fazendo e o que não deve. Fazendo-se o uso de uma medida de desempenho, apresenta-se o exemplo de Russel (2004):

Considerando um aspirador de pó como agente que procura e limpa a sujeira do chão sozinho, poderia-se propor que a medida de desempenho fosse a quantidade de sujeira que foi aspirada em um período de oito horas. No caso de um agente racional, obtém-se o que foi pedido, porém, com o intuito de melhorar esta medida de desempenho, o mesmo pode aspirar a sujeira e logo em seguida despejar novamente ao chão, fazendo isso durante as oito horas. Deste modo o agente teria uma pontuação alta, pois mesmo não limpando a casa inteira, recolheu uma grande quantidade de lixo no período programado.

Um modo mais apropriado de avaliação seria recompensar o agente por cada quadrado limpo do chão, podendo até ter uma penalidade pela eletricidade consumida e o ruído feito. Assim, o aspirador de pó desempenharia realmente a função desejada pelo usuário, que é a de limpar o máximo de sujeira dentro das oitos horas estabelecidas.

Na implementação do agente inteligente que será mostrado no presente trabalho, a categoria adotada para ele será a de um sistema que age, ou atua, racionalmente.

Segundo Russel (2004), o agente racional busca sempre alcançar o melhor resultado ou se há incertezas, o melhor resultado por ele esperado. Porém, para diferenciá-lo de um simples programa é necessário que o mesmo tenha alguns atributos, que possam distingui-los de outros, como ter autonomia, reconhecer o ambiente e adaptar-se ao mesmo, ou segundo Braga (2001), flexibilidade.

Para entender melhor os atributos de um agente, é necessário saber suas características e conhecimento que, segundo Braga (2001), é classificado em três tipos:

- Ambiente: é a capacidade do agente receber entradas, ou como chamado antes, sequência de percepções e executar ações da melhor forma. Em Inteligência Artificial (Russel, 2004), os autores chamam esse conhecimento do ambiente de racionalidade, onde a sua definição está dependente de quatro fatores, que são: “a medida de desempenho que define o sucesso”, “o conhecimento anterior do ambiente que o agente tem do ambiente”, “as ações que o agente pode executar” e “a sequência de percepções do agente até o momento”.

- Flexibilidade: é o que diferencia um programa de controle de processos de um agente inteligente, onde o mesmo deve ser reativo (perceber mudanças em seu ambiente e reagir a elas, oportunamente), proativo (em alguns momentos deve ter iniciativa, visando a resolução do problema) e social (onde o agente deve interagir com outros agentes,

de modo a trocar experiências, para obter a resolução do problema de maneira mais fácil). A flexibilidade é chamada de aprendizado por Russel (2004).

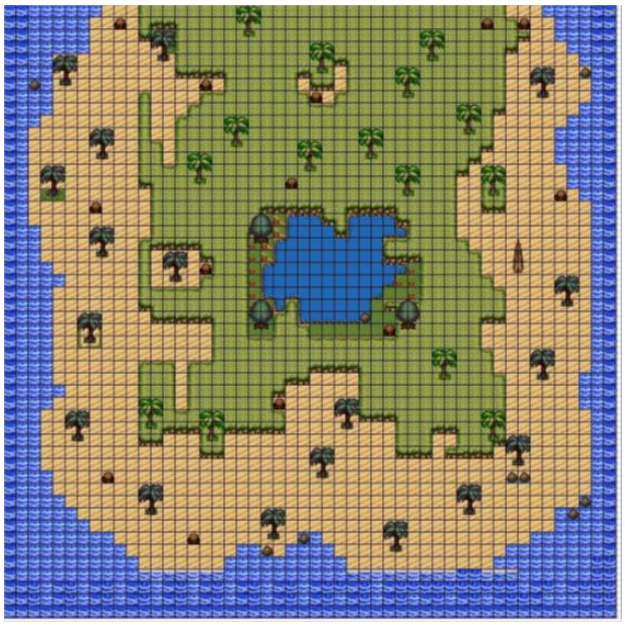
- Autonomia: significa que o sistema executa as tarefas sozinho, sem o intermédio do usuário, ou seja, tem o controle de sua ações e estado interno. Completando a ideia de Braga (2001), Russel (2004), falam que no início, pelo motivo do agente ter pouca experiência, a tendência é do mesmo ter pouca autonomia e agir mais ao acaso, a menos que a pessoa que o projetou tenha dado alguma assistência. Porém, com o passar do tempo e o aprendizado do ambiente, ele começa a ficar mais independente.

2.1.2.1 Exemplo de Aplicação da Técnica

Para exemplificar o que foi explicado sobre agentes inteligentes (Oliveira, 2010), onde o agente, nesse caso, está sozinho e tem que se adaptar a viver em uma ilha, como mostrada na figura a seguir, onde a medida de desempenho do personagem é avaliada pela habilidade do mesmo em manter suas necessidades em um nível adequado, ser feliz e sobreviver.

Para cumprir o objetivo ele apresenta as seguintes funções: andar, comer e excretar, assim como apresenta alguns sensores, visão, paladar e tato. No caso do paladar, foi implementado um grau de desejo, pois haverá vários alimentos na ilha e o personagem pode preferir comer uma banana, que tem o grau de desejo maior do que uma maçã, por exemplo, caso ele tenha os dois a sua escolha. O jogo se passa em um ambiente estático, significando que apenas o agente pode mudá-lo.

Figura 8 – Mapa do simulador



Fonte: OLIVEIRA, 2010

No início da simulação, como apresentado por Oliveira (2010), no mapa estavam espalhados, de maneira aleatória, dez tipos diferentes de comida e de bebida, sendo que vinte por cento de cada uma apresentava uma característica desagradável ao tato ou ao paladar.

O agente, chamado de Brian, no começo não tinha nenhum conhecimento do ambiente, logo, não sabia onde encontrar comida e água. Porém, ao sentir fome e sede começou a explorar o ambiente em busca dos mesmos. Comia o que aparecesse primeiro a sua frente, sem distinção, mas com o passar do tempo, notou-se uma certa personalidade na personagem, que selecionava, mediante ao seu grau de fome, apenas os alimentos que achava mais saborosos.

Como exemplo, no caso de estar com pouca fome e ele passar perto de algo que não goste tanto, o mesmo era ignorado, porém, se estava com muita fome, se alimentava com o que estivesse perto.

Concluindo, Brian se comportou da maneira esperada, onde ele aprendeu com seus erros e conseguiu desenvolver a habilidade de encontrar comida suficiente para suprir suas necessidades. A seguir, pode-se verificar uma imagem do simulador funcionando, onde o agente está a procura de alimento:

Figura 9 – *Brian* procurando alimento

Fonte: OLIVEIRA, 2010

2.1.3 Ferramentas Utilizadas

Para o desenvolvimento do simulador realizado neste trabalho, são usadas duas ferramentas, a linguagem C++ e a API OpenGL.

2.1.4 C++

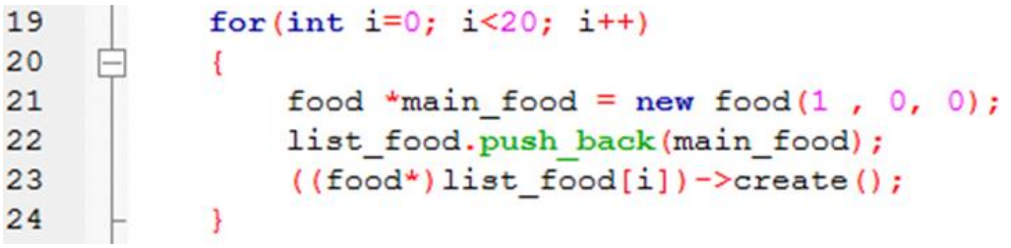
Desenvolvida em 1980, por Dr. Bjarne Stroustrup, é uma linguagem de programação orientada à objeto derivada de C (Johann, 2013).

Para entender melhor sobre C++, a seguir é feita uma breve explicação de algumas estruturas comumente usadas em uma linguagem orientada à objeto, segundo Ricarte (2014):

- Classes: uma classe é um gabarito para a definição dos objetos, ou seja, é nela que o objeto baseia seu comportamento (métodos) e seus atributos. Geralmente, para se expressar uma classe e seus inter-relacionamentos, usa-se uma linguagem de modelagem, como a *UML*.

- Objetos: também conhecidos como instâncias, recebem os atributos e os métodos de uma classe. Segue um exemplo:

Figura 10 – Exemplo de Objeto

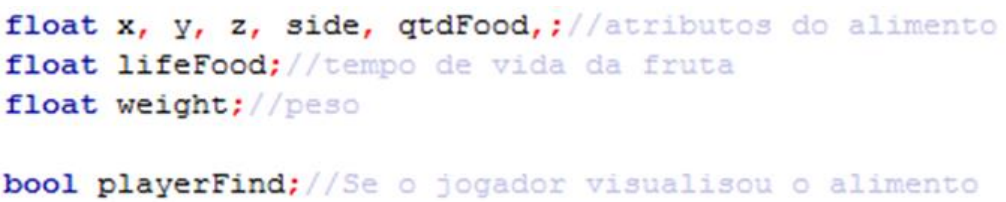


Fonte: Elaborada pelo autor

Neste caso, estão sendo instanciados vinte alimentos, dentro da lista de alimentos, ou seja, são vinte objetos que podem ser diferentes entre eles, mas tendo característica de uma mesma classe.

- Atributos: são o conjunto de propriedades da classe, podem ser chamadas também de variáveis, cada uma apresenta um tipo (inteiro, caractere, booleano...) e um nome associado. Segue um exemplo, da classe *food*:

Figura 11 – Exemplo de Atributos



Fonte: Elaborada pelo autor

- Métodos: também chamada de função, descreve o modo que irão se comportar os objetos daquela mesma classe. Assim como os atributos, os métodos também apresentam um nome (neste caso chamado de assinatura) e um tipo para o valor do retorno. Um exemplo é mostrado na figura a seguir, da classe *GeneticAlgorithm*, do código do desenvolvimento do simulador:

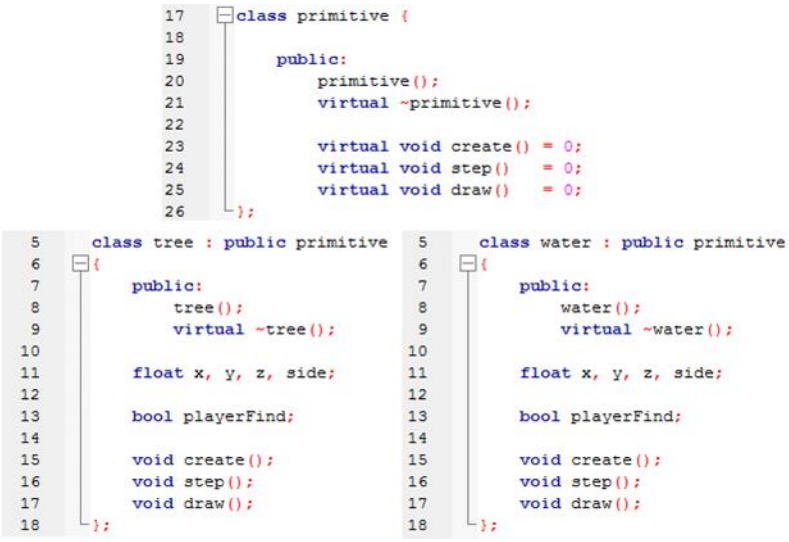
Figura 12 – Exemplo de Métodos

```
40      void raffleAttributes();//sorteio dos atributos do player
41      void fitnessPlayers();//2 melhores player para serem escolhidos para o cruzamento
42      void cross();//cruzamento
43      int mutation(int, int, int);//chance de fazer a mutação
```

Fonte: Elaborada pelo autor

- Heranças: é um mecanismo que permite que características (os métodos e os atributos) sejam passadas da classe base para as subclasses. Deste modo, cada classe filha terá as características da classe pai, que são mais gerais e as especificas de sua própria classe. Para exemplificar o que foi dito, considere a classe pai como `Animal`, onde temos alguns atributos como peso, força e comprimento, e o método `EmitirSom` que são comuns em todos os animais da classe filho. Temos as classes `Gato` e `Cao` como filhos, porém um emite um som de miado e o outro de latido, ou seja, quando a função `EmitirSom` do pai for evocada, será tratada para que cada um dos filhos tenham seus respectivos sons emitidos. Segue um exemplo retirado do simulador:

Figura 13 – Exemplo de Herança



Fonte: Elaborada pelo autor

Neste caso, a classe `primitive` é pai de todas as outras classes do simulador, mas para exemplificar, apenas foram mostradas as classes `tree` e `water` como filhas, que contém os métodos `create`, `step` e `draw`, implementados diferentemente.

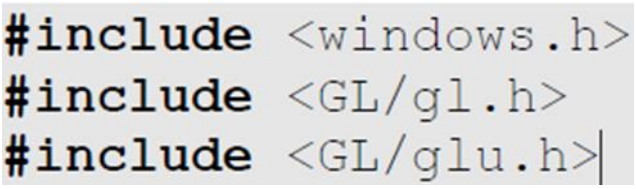
2.1.4.1 OpenGL

Definida como uma API, ou biblioteca de interface gráfica, portátil, para criação de aplicações gráficas bidimensionais (2D) e tridimensionais (3D), segundo Manssour (2014). O OpenGL foi criado para ser usado em várias plataformas e sistemas operacionais, assim como para diversas linguagens e ambientes de programação (Valente, 2014).

No caso do sistema operacional *Windows*, como dito por Valente (2014), “OpenGL já está pronta para uso (em modo simulado, por *software*, pelo menos). A implementação da biblioteca está contida em arquivos DLL (opengl32.dll e glu32.dll)”.

Para programar uma aplicação usando OpenGl é necessário incluir os seguintes arquivos:

Figura 14 – Arquivos do OpenGL



Fonte: VALENTE, 2014.

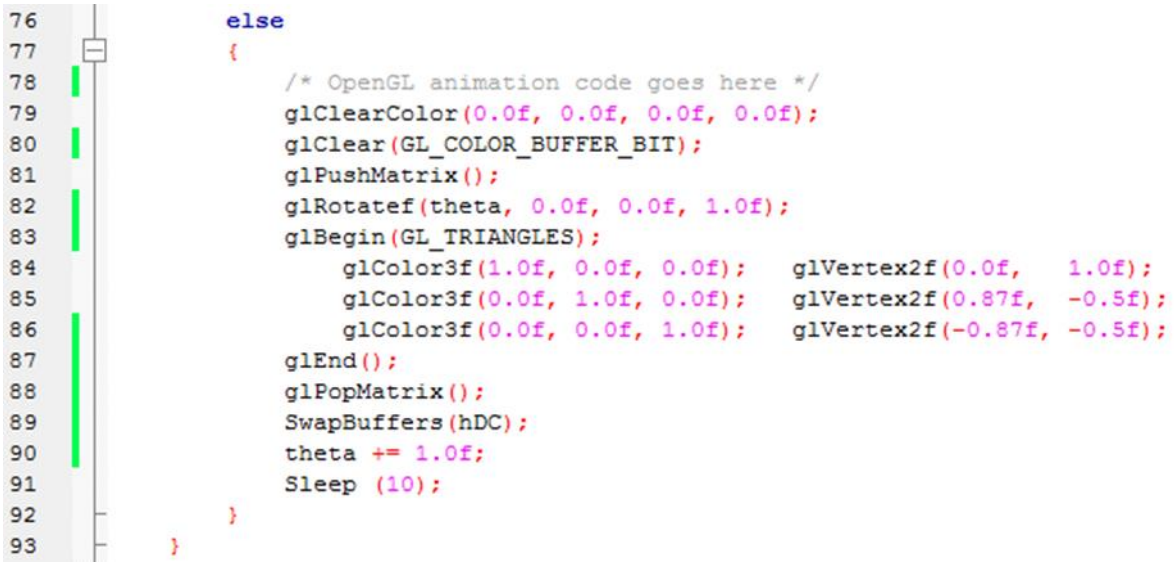
O uso OpenGL se deu pelo fato do próprio já tratar das *thread*, que segundo CPLUSPLUS (2014), é uma sequência de instruções que podem ser executadas simultaneamente, gerando um código principal pronto para começar o desenvolvimento.

Através da função gráfica *double buffering* do OpenGL, é oferecida uma animação mais suavizada dos objetos, diferente do *flickering* ou cintilação[1]. Segundo Sobrinho (2003), essa suavidade acontece, pois a cena que será mostrada em sequência já esta sendo construída em *background*, então só depois é mostrada, permitindo que apenas as imagens completas sejam mostradas na tela.

A função que torna isso possível é a `SwapBuffer`, recebendo um parâmetro do tipo `HDC`, que é a cena que está sendo construída em *background*, retornando verdadeiro se obteve sucesso e falso se falhou, logo depois mostrando uma mensagem de erro. Segue o exemplo do uso da função, com o código que é gerado automaticamente :

[1] Agitação rápida de uma luz que brilha e se ofusca continuamente.

Figura 15 – Exemplo do uso do *SwapBuffer*



```
76          else
77          {
78              /* OpenGL animation code goes here */
79              glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
80              glClear(GL_COLOR_BUFFER_BIT);
81              glPushMatrix();
82              glRotatef(theta, 0.0f, 0.0f, 1.0f);
83              glBegin(GL_TRIANGLES);
84                  glColor3f(1.0f, 0.0f, 0.0f);   glVertex2f(0.0f,   1.0f);
85                  glColor3f(0.0f, 1.0f, 0.0f);   glVertex2f(0.87f,  -0.5f);
86                  glColor3f(0.0f, 0.0f, 1.0f);   glVertex2f(-0.87f, -0.5f);
87              glEnd();
88              glPopMatrix();
89              SwapBuffers(hDC);
90              theta += 1.0f;
91              Sleep (10);
92          }
93      }
```

Fonte: Elaborada pelo autor

Assim que inicia o programa é habilitado o OpenGL com a função `EnableOpenGL`, depois entra no *loop* principal, o qual está sendo mostrado na figura 15 e ao final, na linha oitenta e nove, aparece o `SwapBuffer`, que faz a tela que estava no *background* ser apresentada na a tela principal, tendo, nesse caso dez milissegundos, `Sleep(10)`, na linha noventa, para criar a tela de *background* novamente. Repetindo este processo até que seja apertada a tecla ESC.

3 Proposta de trabalho

A proposta de trabalho busca o desenvolvimento de um simulador de sobrevivência, onde um agente inteligente é deixado em uma ilha para aprender a sobreviver ao ambiente.

No início ele não sabe muito bem o que pode fazer e a melhor hora de fazer algo, sendo movido pelos instintos básicos, como conhecer o ambiente, comer, dormir e tomar água, tendo todos seus atributos sorteados. Porém, com o tempo irá se adaptando e descobrindo as melhores maneiras de agir no ambiente.

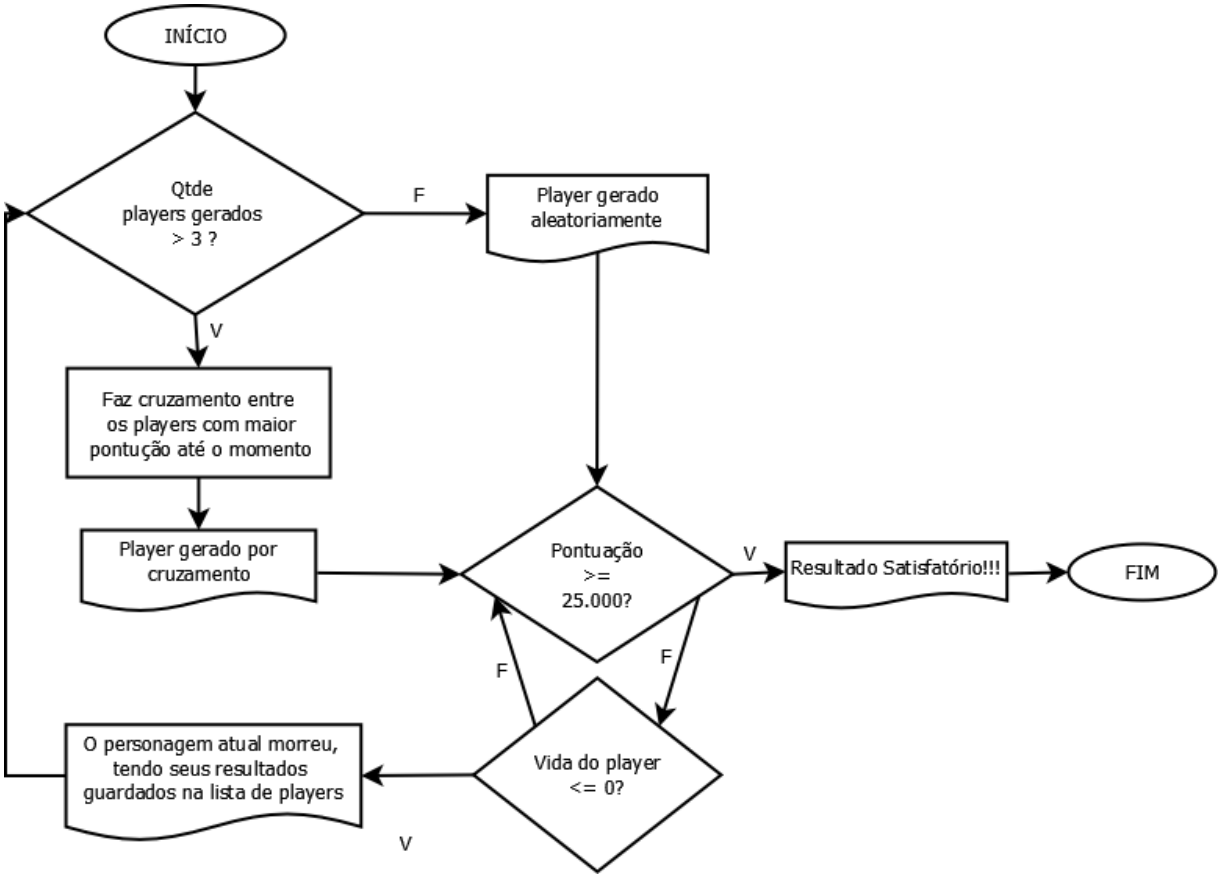
No decorrer da simulação, o agente inteligente vai evoluindo, tanto dentro da própria experiência, pois terá apenas um personagem por vez no ambiente, como pela sua descendência que é guardada em uma lista, porque, após a morte de um dos agentes, são escolhidos os dois seres que melhor se adaptaram ao ambiente, cruzando-os, usando o algoritmo genético para obter um descendente mais evoluído.

O simulador faz esse ciclo até que a personagem alcance a pontuação de vinte e cinco mil, que será considerada como um resultado satisfatório.

O valor de vinte e cinco mil foi escolhido após vários testes realizados no simulador, verificando que ao alcançar esta pontuação, o *player* não iria morrer.

Para ilustrar o funcionamento do programa, o mesmo foi apresentado por um fluxograma, como mostrado a seguir:

Figura 16 – Fluxograma do simulador que será apresentado neste trabalho



Fonte: Elaborada pelo autor

4 Desenvolvimento do Jogo

Neste capítulo apresenta-se o jogo proposto no tema do presente trabalho, primeiramente fazendo um resumo e logo após, detalhando melhor o objetivo esperado para o término do desenvolvimento.

Também é apresentado o funcionamento do jogo, explicando melhor sobre a IA implementada no agente inteligente, os objetos que são encontrados no ambiente que a personagem pode interagir, o detalhamento da pontuação de cada ação tomada e como o algoritmo genético está sendo usado, sempre mostrando trechos de código, conforme for necessário.

4.1 Resumo do Jogo

O jogo é um simulador de sobrevivência, onde o agente inteligente tem que se adaptar ao ambiente, que lhe fornece uma quantidade limitada de alimento, no caso, frutas, que delas, podem ser retiradas sementes para plantar uma nova fruta, alguns lagos para saciar a sede e levar água para regar as plantas, e árvores para usar os pedaços de madeira para fazer uma casa.

Para ajudar o agente nas escolhas a serem tomadas durante a simulação, foram colocados atributos como fome, sede e sono, que conforme suas necessidades faz com que ele tome as melhores decisões para resolver seu problema. O personagem ainda apresenta um atributo vida, que conforme um dos outros atributos (fome, sede e sono) estiverem com o nível menor que trinta porcento, começa a ser retirada uma certa quantidade de vida e caso ela chegue a zero, a personagem será considerada morta.

Após a morte da personagem, será feita a escolha dos dois seres, dentro da lista de *players*, mais aptos ao ambiente através de sua pontuação, utilizando o algoritmo genético para cruzá-los, na intenção de o descendente obter um resultado melhor, se adaptando de maneira mais eficiente que seus antecessores, isto será feito, até o agente chegar a um resultado satisfatório.

4.2 Objetivo do Jogo

O objetivo do desenvolvimento do simulador é que a cada geração de descendentes, a pontuação dos mesmos aumente, de maneira que eles se adequem melhor ao ambiente, resistindo mais. Para isso ser possível, é necessário o uso de atributos (vida, fome, sede e sono) e aprendizados (conhecimento para plantar e construir a casa).

No começo da simulação, os atributos são aleatórios e seu aprendizado é baixo, ou seja, não sabe como e onde plantar e fazer sua casa, para ter um descanso mais rápido.

O que é esperado como resultado satisfatório é o agente, após alguns cruzamentos realizados, aprender a plantar (saber a melhor hora de regar a semente e o melhor lugar para plantá-la), construir uma casa próxima a um lago e que toda sua plantação esteja perto delas (casa e lago).

Desta forma, a personagem deve explorar o ambiente para descobrir onde tem uma árvore, para extrair madeira, e um lago, para fazer uma casa por perto. Nessa exploração o agente deve, conforme se alimenta, recolher as sementes deixadas pelas frutas, para posteriormente, plantá-las perto de sua casa. Com isso, o agente terá tudo do que necessita próximo de sua casa, diminuindo esforços para pegar água, regar as sementes, dormir e se alimentar.

4.3 Funcionamento do Jogo

Neste capítulo é explicado de maneira mais detalhada o funcionamento do jogo, mostrando trechos de código quando for necessário, para obter um melhor entendimento da personagem principal (agente inteligente), de todos os objetos que o mesmo pode ter acesso no ambiente, a pontuação de suas ações, de modo a ter o controle dos melhores agentes e como o Algoritmo Genético foi implementado no simulador.

4.3.1 Objetos Encontrados no Ambiente

O agente tem a seu dispor alguns diferentes tipos de objetos para sua sobrevivência e cada personagem terá um ambiente distinto, porém, sempre com a mesma quantidade inicial de frutas, árvores e lagos.

Ao nascer, o *player* começa a explorar o ambiente para conhecê-lo melhor, pois no começo não sabe onde estão os objetos, representados com uma cor mais escura, e precisa achá-los, para memorizar sua posição e voltar nele posteriormente, conforme necessitar. Tudo que a personagem já encontrou no cenário está com a sua cor mais clara, comparado com a cor de um objeto que ainda não foi encontrado. Segue a imagem com o *player*, usando sua percepção do ambiente (circúlo branco):

Figura 17 – Agente

Fonte: Elaborada pelo autor

Percebe-se que o que está dentro de sua percepção está mais claro, ou seja, já foi visto e poderá ser acessado assim que precisar e o objeto que está fora tem sua cor mais escura, indicando que ainda não foi visto.

A seguir, uma explicação mais detalhada de todos objetos que poderão ser encontrados pelo agente:

- Alimentos: são considerados como frutas, de modo que após se alimentar, com a semente caindo no chão, sem que o jogador perceba, ou quando já tiver consciência que nestas frutas contém sementes, o mesmo recolhe-as

para plantá-las. Representados por um quadrado vermelho, alimentar-se delas recupera a barra de *Hungry*.

- Lago: no simulador são mostrados como um quadrado azul, tem duas funções, a de saciar a sede do personagem, que é a barra *Water*, e a de encher seu copo, para regar as sementes, que é representado pelo *Water inCup*.

- Árvores: servem para se retirar madeira, para futuramente construir uma casa. Para construir a mesma, é necessário ter uma quantidade de madeira igual a dez no marcador, o *Qtd Wood*. São representadas como um quadrado laranja no simulador.

- Casa: antes construir a casa, o *player* dorme no momento em que sente sono, independentemente do lugar, porém desta maneira, sua recuperação do cansaço é mais lenta, equivalendo a uma noite mal dormida. Por isso a necessidade de ter uma casa, pois assim, o mesmo repousará somente nela, recuperando sua barra de cansaço, *Tired*, muito mais rápido. Segue a figura da casa:

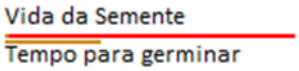
Figura 18 – Casa

Fonte: Elaborada pelo autor

Porém, após recolher a quantidade de madeira suficiente, é preciso construir a casa, nesta imagem mostra-se a mesma no momento de sua construção, onde a porcentagem de tempo que ainda falta é representado pela barra vermelha.

- Semente: o objetivo é germinar, para nascer outra fruta dela, porém, o agente deverá ter conhecimento do melhor momento para se regar a semente para que ela não morra. Segue a figura como exemplo:

Figura 19 – Semente



Fonte: Elaborada pelo autor

Quando uma semente é plantada, junto à ela há uma barra de vida e do tempo para germinar, vermelha e amarela respectivamente, se a vida for baixa, a velocidade de germinar diminui, se a vida for zero, a semente morre, mas, se o tempo de germinar for igual a zero, significa que a semente se tornou uma nova fruta.

4.3.2 Forma que será utilizado o Algoritmo Genético

Como mostrado na figura 16, no fluxograma do funcionamento do simulador, os três primeiros players têm seus atributos, *maxHungry*, *maxWater*, *maxTired* e *learnPlant*, gerados de maneira aleatória na função `raffleAttributes` da classe `GeneticAlgorithm`, onde os três primeiros atributos têm seu escopo entre dez e noventa, e o ultimo, entre zero e cinquenta.

Figura 20 – Função *raffleAttributes*

```
332   void GeneticAlgorithm::raffleAttributes()
333   {
334       maxHungry = rand()%80 + 10;
335       maxTired = rand()%80 + 10;
336       maxWater = rand()%80 + 10;
337       learnPlant = rand()%50;
338   }
```

Fonte: Elaborada pelo autor

Após estes primeiros agentes serem gerados e testados no ambiente de simulação, tem-se parâmetros para começar a gerar as próximas gerações por

cruzamento. Para isso, são escolhidos os dois melhores *players*, pela sua pontuação final no jogo.

Primeiramente, na função `fitnessPlayers`, é realizado um laço de repetição na lista de jogadores, para buscar os dois que tiveram mais pontuação.

Em seguida, na função *cross*, os valores dos atributos dos *players* escolhidos, são usados como os valores máximos e mínimos do escopo do sorteio do valor do atributo do *player* da próxima geração. Por exemplo, no caso do `maxHungry`, os melhores agentes tem os seguintes números, trinta e três e cinquenta, ou seja, o próximo ser terá seu `maxHungry` entre estes valores, caso não ocorra a mutação. Para ilustrar melhor, seguem trechos do código da função em questão:

Figura 21 – Trecho da função *cross*

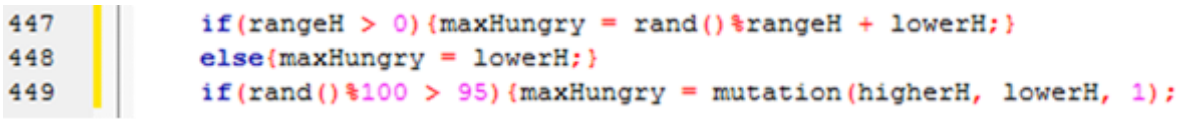
```
375     int lowerH, higherH, rangeH;//menor, maior e maior - menor, valor de Hungry
376     int lowerT, higherT, rangeT;//menor, maior e maior - menor, valor de Tired
377     int lowerW, higherW, rangeW;//menor, maior e maior - menor, valor de Water
378     int lowerP, higherP, rangeP;//menor, maior e maior - menor, valor de learnPlant
380    //faz o rangeA
381    if( ((player*)list_player[p1])->maxHungry > ((player*)list_player[p2])->maxHungry )
382    {
383        higherH = ((player*)list_player[p1])->maxHungry;
384        lowerH = ((player*)list_player[p2])->maxHungry;
385    }
386    else
387    {
388        higherH = ((player*)list_player[p2])->maxHungry;
389        lowerH = ((player*)list_player[p1])->maxHungry;
390    }
391    rangeH = higherH - lowerH;
```

Fonte: Elaborada pelo autor

Nestes trechos são mostradas as variáveis `lower`, `higher`, `range`, para cada atributo, que significam, respectivamente, o menor valor, o maior valor e o escopo. Em seguida é mostrado o exemplo da atribuição destas variáveis, se caso o *player* `p1` tiver o maior valor do `maxHungry` do que o `p2`, é atribuído o valor do atributo do `p1` no `higherH` e a do `p2` no `lowerH`, se não, `p1` será igual a `lowerH` e `p2` será igual a `higherH`. No caso do escopo, é igual a `higherH` menos `lowerH`, como mostrado na linha trezentos e noventa e um do código.

A seguir, o código mostra o cruzamento:

Figura 22 – Cruzamento e mutação

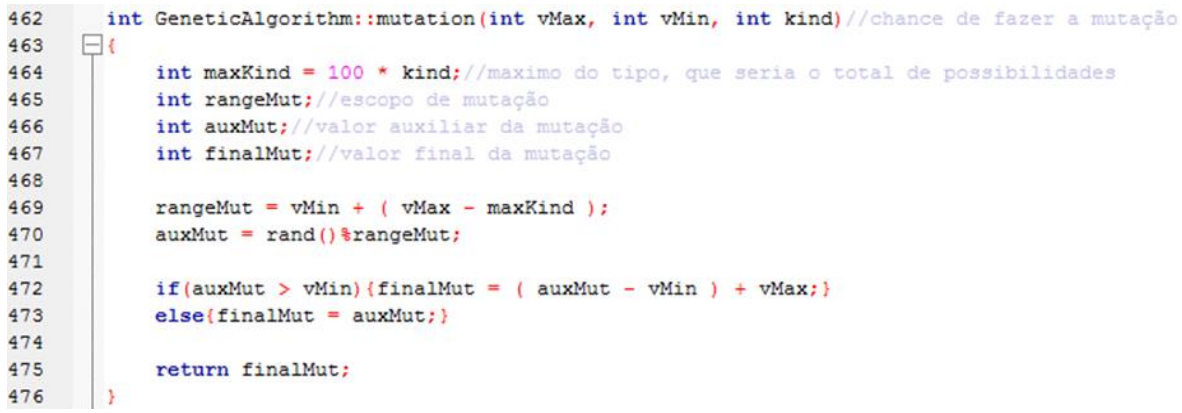


```
447     if(rangeH > 0){maxHungry = rand()%rangeH + lowerH;}
448     else{maxHungry = lowerH;}
449     if(rand()%100 > 95){maxHungry = mutation(higherH, lowerH, 1);
```

Fonte: Elaborada pelo autor

A primeira condição, verifica se o escopo é maior que zero, pois, caso o valor do mesmo seja zero, na hora de fazer o sorteio irá dar erro, se não, caso for zero, o valor do `maxHungry` será igual ao menor valor. Na próxima linha, é feito um sorteio de zero a cem, onde se tem cinco por cento de chance de acontecer a mutação, mas caso ela ocorra, a função `mutation` será chamada. A implementação desta função será mostrada a seguir:

Figura 23 – Trecho da função *mutation*



```
462   int GeneticAlgorithm::mutation(int vMax, int vMin, int kind)//chance de fazer a mutação
463   {
464       int maxKind = 100 * kind;//maximo do tipo, que seria o total de possibilidades
465       int rangeMut;//escopo de mutação
466       int auxMut;//valor auxiliar da mutação
467       int finalMut;//valor final da mutação
468
469       rangeMut = vMin + ( vMax - maxKind );
470       auxMut = rand()%rangeMut;
471
472       if(auxMut > vMin){finalMut = ( auxMut - vMin ) + vMax;}
473       else{finalMut = auxMut;}
474
475       return finalMut;
476   }
```

Fonte: Elaborada pelo autor

Primeiramente, são passados para a função três valores: o valor máximo, o valor mínimo e o tipo, ou `kind`. Sobre os dois primeiros, eles são os valores máximos e mínimos de um dos atributos, e o `kind` é o tipo de atributo que foi usado no código apenas para facilitar na hora de atribuir valor a variável `maxKind`, pois no caso dos atributos `hungry`, `tired` e `water` o valor máximo é igual a cem, ou seja o

valor de `kind` será igual a um, mas se o atributo que for sofrer a mutação for o `learnPlant`, então o valor do tipo será de dois, pois o total do mesmo é de duzentos.

A ideia da mutação é que se tenha um valor diferente do que se esperava, se o cruzamento fosse normal. Desta maneira, se o cruzamento não tivesse mutação, o valor do atributo seria entre o valor mínimo e o valor máximo, mas com a mutação, o valor esperado será entre zero e o valor mínimo ou entre o valor máximo e o `maxKind`.

Concluindo, deve-se salientar que todo o processo teve como base o pseudocódigo apresentado anteriormente por Russel (2004), com uma diferença no cruzamento, onde acontece o *crossover*. Segundo Russel (2004), o novo indivíduo deve ter uma subcadeia de genes do pai e outra da mãe. No entanto, no presente trabalho, o cruzamento é feito gene a gene, no caso, são os atributos, como se em todos os genes tivessem uma contribuição do pai e da mãe, não apenas copiando um bloco de genes de cada.

4.3.3 Pontuação

Conforme afirmado anteriormente por Russel (2004), para se avaliar o desempenho de qualquer agente, é preciso fazer uma pontuação de suas ações, as boas e as ruins. Com base nisso foram feitas as tabelas seguintes, pontuando todas as ações que o personagem pode fazer durante a simulação.

Esta primeira tabela mostra a pontuação feita a cada atualização do jogo, onde os valores são incrementados ou decrementados a seu valor anterior, dependendo de como estão seus atributos. A pontuação `sSurvival`, apenas tem o incremento, pois é um bônus ao tempo que o agente sobreviveu no ambiente. Segue a tabela:

Tabela 1 – Pontuação modificada sempre que se atualiza o jogo

Pontuação modificada sempre que se atualiza o jogo			
Escopo	**Hungry**	**Water**	**Sleep**
Maior que 70%	+ 0,03	+ 0,02	+ 0,01
Entre 70% e 50%	+ 0,015	+ 0,01	+ 0,007
Entre 50% e 30%	+ 0,005	+ 0,00	+ 0,005
Entre 30% e 10%	- 0,01	- 0,01	+ 0,00
Menor que 10%	- 0,05	- 0,04	- 0,1
Survival	+ 0,01		

Fonte: Elaborada pelo autor

A pontuação para a fome é mais alta, pois ela é considerada o atributo mais importante, porém o sono apresenta o maior decremento, porque, com sua porcentagem menor que dez por cento, sua velocidade é muito baixa e sua área de visão é muito reduzida.

A seguir, a tabela dos pontos referentes a pontuação com a madeira, tanto as que são atualizadas a cada momento que é realizada a ação, como as recompensas por cumprir uma tarefa, onde este incremento é feito apenas uma vez durante a simulação:

Tabela 2 – Pontuação da madeira

Pontuação da madeira	
A cada atualização que o agente...	
Cortar Árvore	+ 0,005
Construir Casa	+ 0,1
Bônus por tarefa feita...	
10 Toras de Madeira	+ 100,00
Fazer Casa	+ 100,00

Fonte: Elaborada pelo autor

A Tabela 3 é a de plantação. Seu quesito de avaliação é a distância entre a semente plantada e o rio mais próximo, levando em consideração se o agente sabe

ou não plantar. No caso dele não saber plantar seria como se ele comesse a fruta e sem saber deixasse cair a semente no chão, neste caso a pontuação será setenta por cento menor do que no caso do agente que sabe plantar.

Porém, conforme o *player* tiver mais conhecimento de plantação ao final da simulação, a pontuação da plantação será multiplicada. A seguir a tabela:

Tabela 3 – Pontuação da plantação

Pontuação da Plantação		
Por Proximidade de algum lago		
	Não Sabe Plantar	Sabe Plantar
Até 80px	+ 30,00	+ 100,00
Entre 80px e 140px	+ 18,00	+ 60,00
Entre 140px e 200px	+ 3,00	+ 10,00
Entre 200px e 300px	+ 0,00	+ 0,00
Acima de 300px	- 9,00	- 30,00

Fonte: Elaborada pelo autor

Após mostrada todas as tabelas, a pontuação final se dá da seguinte maneira, sem dar um peso maior a nenhuma das pontuações:

Figura 24 – *Score*

```
262         score = sSleep + sWater + sHungry + sPlant + sWood + sSurvival;
```

Fonte: Elaborada pelo autor

4.3.4 Personagem Principal

O Agente Inteligente ou a personagem principal tem como objetivo manter seus atributos sempre em níveis considerados, para ele, os mais adequados, dando sempre prioridade à sua alimentação, sede e sono, consecutivamente. Para ele, foi implementada a classe `player` e uma lista contendo todos os *players* que já participaram da simulação (`list_player`).

No `header` da classe `player`, para facilitar a manipulação das ações do jogador, foi criado um `enum`[1], com cada ação que o personagem faz durante o jogo:

Figura 25 – *Status* do agente

```
16    enum actionPlayer
17    {EATING, GOTOFOOD, SLEEPING, GOTOSLEEP, STOP, SEARCH, GOTOWATER,
18    DRINKING, HELPSEED, GETWATER, GOTOSEED, GOTOPLANT, GOTOGETSEED,
19    GOTOTREE, CUTTREE, GOTOBUILD, BUILDHOUSE};
```

Fonte: Elaborada pelo autor

As ações `GOTO` são para ir à algum lugar, são os estados quando se está caminhando para um objeto, como por exemplo comida ou casa (para dormir ou para construí-la). Sempre antes de entrar em estado `GOTO`, o mesmo verifica o objeto mais próximo do tipo desejado (comida, água ou árvore), dentro daqueles que já foram encontrados, para ir até ele. Existem algumas exceções, o `GOTOSEED`, o `GOTOPLANT`, o `GOTOBUILD` e o `GOTOSLEEP`.

No caso do `GOTOSEED`, é feita uma função para avaliar a condição de todas as sementes (segundo as expectativas do jogador) e verificar a distância em que se encontra em relação a mesma, se ele consegue chegar a tempo de regar a semente para ela não morrer, ou seja, o *player* tenta salvar primeiro as sementes que estão em um estado mais crítico, o ato de cuidar da mesma é o `HELPSEED`.

Em `GOTOPLANT`, a personagem, conforme seus conhecimentos de plantação ou `learnPlant`, vai plantar as sementes que tem guardada.

Nos dois outros casos, pelas ações estarem vinculadas a casa, que sempre terá apenas uma, seu `GOTO` irá apontar sempre para o mesmo lugar, sem ter a necessidade de fazer cálculos de distância.

Deve-se explicar melhor algumas variáveis usadas:

[1] Constantes do tipo inteiro nomeadas.

- `float status`: é o estado, ou ação atual que o personagem está realizando.

- `float learnPlant`: é o conhecimento de plantação que o player apresenta, tem como valor mínimo, zero, e valor máximo, duzentos, onde após os cem pontos, já se tem a noção de plantação.

- `float velTired`: é a velocidade do jogador, levando em consideração seu cansaço, este valor vai de 0,05 à 1, resultando na seguinte formula da sua velocidade final, `vel = maxVel * velTired`, onde o `maxVel` vale 2,5. Segue a tabela ilustrando a situação:

Tabela 4 – Velocidade com o cansaço

Velocidade com o cansaço	
Escopo	**velTired**
Maior que 70%	1
Entre 70% e 50%	0,7
Entre 50% e 30%	0,5
Entre 30% e 10%	0,3
Menor que 10%	0,05

Fonte: Elaborada pelo autor

- `float effort`: é atribuído a essa variável, o valor do esforço do *player* ao realizar uma tarefa, de modo que as tarefas que requerem mais força bruta, gastem mais energia, ou seja cansem o jogador mais rápido, seguindo a lógica da formula `sleep -= 0.01 * effort`. A seguir, a função `effortStatus`, para mostrar os valores do effort para cada status:

Figura 26 – Classe *effortStatus*

```
1737  void player::effortStatus()
1738  {
1739      if(status == CUTTREE || status == BUILDHOUSE){effort = 2.0;}
1740
1741      if(status == GOTOFOOD || status == GOTOSLEEP || status == GOTOWATER ||
1742         status == GOTOSEED || status == GOTOPLANT || status == GOTOGETSEED ||
1743         status == GOTOTREE || status == GOTOBUILD || status == SEARCH)
1744         {effort = 1;}
1745
1746      if(status == HELPSEED || status == GETWATER){effort = 1.0;}
1747
1748      if(status == EATING || status == DRINKING){effort = 0.5;}
1749
1750      if(status == STOP){effort = 0.3;}
1751
1752      if(status == SLEEPING){effort = 0.1;}
1753  }
```

Fonte: Elaborada pelo autor

- `bool explorerLvl1`: foi feita essa variável para verificar se o mínimo de exploração do ambiente já foi feito. É considerada como verdadeira quando já foi encontrado um lago, uma árvore e setenta por cento do total de alimentos.

Também é importante salientar o que fazem as funções do `player`, segue a lista das mesmas:

Figura 27 – Funções do agente

```
 97 void tired();
 98 int analyzeFood();
 99 int analyzeWater();
100 void gotoFood(int);
101 void gotoWater(int);
102 void sleeping();
103 void explorer();
104 bool findWater();
105 int analyzeSeed();
106 void gotoSeed(int);
107 void updateScore();
108 void planting();
109 void placePlant(float);
110 void gotoPlant();
111 int getSeed(int);
112 void gotoGetSeed(int);
113 int analyzeWood();
114 void gotoWood(int);
115 void placeBuild(float);
116 void gotoBuild();
117 void effortStatus();
```

Fonte: Elaborada pelo autor

As funções `analyze()`, com retorno de um número inteiro, indicam qual objeto deve ser o alvo do agente, pela sua posição na lista, que logo em seguida são chamadas funções sem retorno, `goto`, que recebem um inteiro, o resultado da função `analyze`, que são para ir em direção do objeto passado.

Seguindo, em ordem, serão expostas as demais funções do jogador:

- `tired`: é dado o valor para a variável `velTired` e o raio de visão do jogador, conforme seu cansaço, quanto mais, menor o raio da visão.

- `sleeping`: caso o player esteja precisando dormir é chamada esta função. Caso o mesmo tenha construído a casa, ele segue em direção à ela para dormir, se não, ele dorme por onde estiver mesmo.

- `explorer`: quando o agente está sem nenhuma necessidade e a casa já foi construída, é chamada esta função para ele explorar o ambiente e tentar descobrir novos objetos.

- `findWater`: a função retorna um booleano, verdadeiro se já foi visto um lago e falso se ainda não.

- `updateScore`: é nesta função que se atualiza a pontuação do *player*.

- `planting`: ao ser chamada, o agente planta a semente e já incrementa a pontuação, referente ao lugar plantado.

- `placePlant`: escolhe a distância em que será plantada a semente de um lago, recebendo como parâmetro na função, o nível de aprendizado de plantação do agente, conforme a tabela a seguir:

Tabela 5 – Nível de plantação

Nível de Plantação	
learnPlant	**Nível**
Menor que 120%	0
Entre 120% e 140%	1
Entre 140% e 160%	2
Entre 160% e 180%	3
Maior que 180%	4

Fonte: Elaborada pelo autor

- `getSeed`: quando o agente não tem mais o que fazer, atributos altos, casa construída e sem sementes para cuidar, ao invés de ficar parado, ele escolhe uma fruta e a retorna, levando em consideração o parâmetro passado para função, nível de plantação, para ir até ela (`gotoGetSeed`) e retirar sua semente.

- `placeBuild`: novamente, levando em consideração o nível de plantação, passado para a função, é escolhido o lugar para se construir a casa.

No decorrer do jogo, como já explicado, o *player* dá preferência a sua alimentação, porém, se caso ele já esteja fazendo alguma outra ação, como por exemplo bebendo água, só para de beber caso a fome do agente esteja menor que cinquenta por cento do valor da fome máxima suportada, ou `maxHungry`. Resumindo, o player só muda seu status caso uma ação de maior preferência esteja com seu valor máximo abaixo de cinquenta por cento.

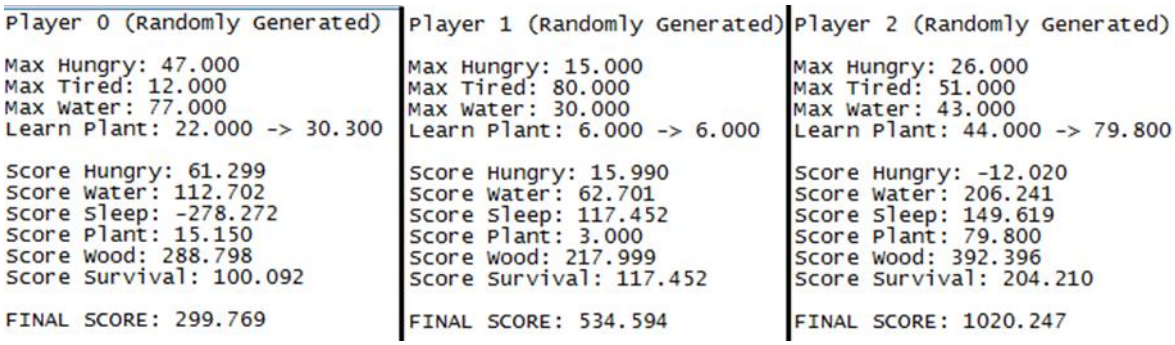
Para recolhimento das toras de madeira e para a construção da casa, o jogador dá a menor preferência, ou seja, só faz essas ações caso não tenha nenhuma outra necessidade, pois é um trabalho que exige muito dele.

4.4 Resultados Obtidos

São apresentados quatro resultados obtidos no simulador, de maneira a ilustrar o funcionamento do algoritmo genético e seus resultados.

Na primeira simulação realizada, seguem os três primeiros seres, gerados de modo aleatório, junto com suas respectivas pontuações:

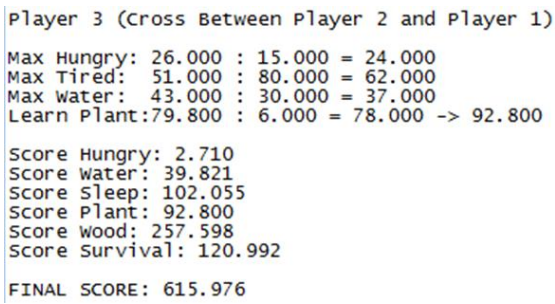
Figura 28 – Primeiro teste, primeiros três agentes



Fonte: Elaborada pelo autor

Neste caso, as personagens escolhidas para o primeiro cruzamento serão a um e a dois, tendo como resultado a personagem três, como mostrado a seguir:

Figura 29 – Primeiro teste, primeiro cruzamento entre agentes

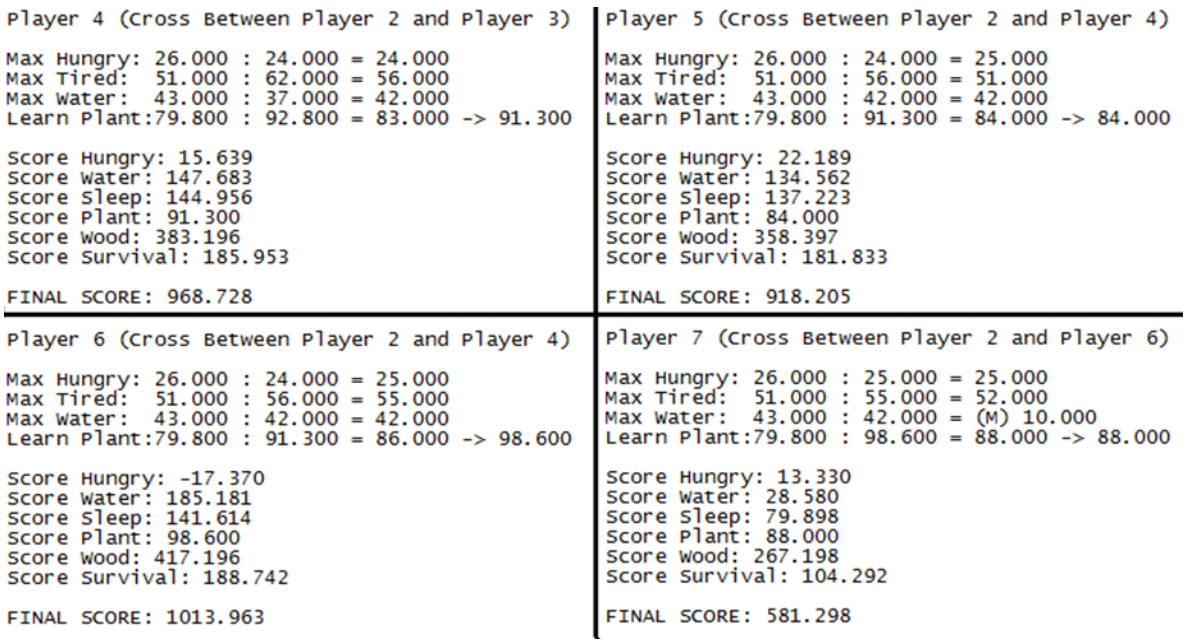


Fonte: Elaborada pelo autor

Na imagem, no caso do *Max Hungry*, por exemplo, o primeiro valor significa, vinte e seis, que é do *player* 2 (o melhor até o momento), e o segundo, quinze, que é do *player* 1(o segundo melhor), tendo como resultado do cruzamento vinte e quatro.

Na sequência, foram gerados mais seres, assim como mostrado na Figura 30:

Figura 30 – Primeiro teste, mais resultados de cruzamentos entre agentes

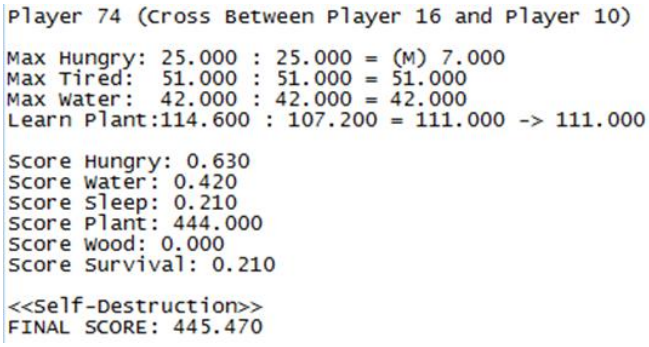


Fonte: Elaborada pelo autor

Verificando estes seres percebe-se que o atributo *Max Hungry* das próximas gerações não passará de trinta, como dito anteriormente, quando um dos valores estiverem menores que trinta, o player começa a perder vida, o que não é ideal para resultado, logo, para se chegar a um valor aceitável, o único modo é que ocorra uma mutação, representada com um (M) antes do atributo do novo *player*, como exemplo de mutação o atributo *Max Water* do *Player 7*, que tem como valor dez.

Porém, a mutação pode ser para um resultado pior, neste caso, um o valor menor do que o já apresentado pelos pais, como o exemplo retirado do primeiro teste:

Figura 31 – Primeiro teste, caso de mutação

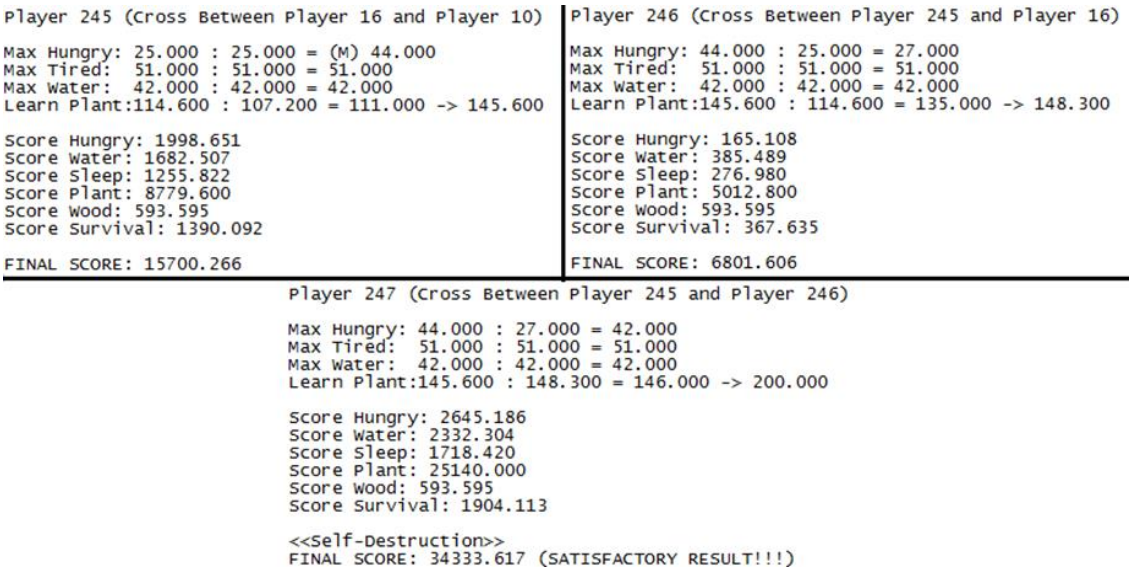


Fonte: Elaborada pelo autor

Com a mutação do atributo do *player* 74, seu valor foi para sete. Sabendo que esse ser teria claramente um resultado pior, comparado com os outros que já foram gerados e entendeu-se que a simulação do mesmo fazia-se desnecessária, autodestruindo o mesmo assim que é iniciada a simulação.

Como a chance de ter uma mutação em um dos atributos é de cinco por cento e ainda precisa-se esperar que a mesma seja uma boa mutação, um valor maior que trinta no atributo *Max Hungry* só foi aparecer no *player* 245, como mostrado a seguir:

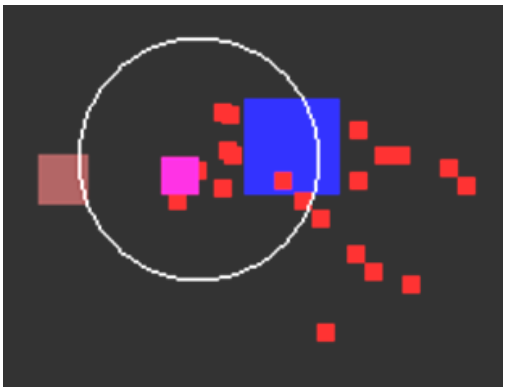
Figura 32 – Primeiro teste, últimos agentes



Fonte: Elaborada pelo autor

Após a morte do *player* 245, sua próxima geração, teve seu atributo com valor inferior à trinta, porém obteve a segunda melhor pontuação até o momento, e, como consequência, o *player* 457 *teria seu Max Hungry* entre quarenta e quatro e vinte e sete, tendo como valor final quarenta e dois. Agora todos os atributos estavam com valores consideráveis como aceitáveis. Como resultado, este ser atingiu um resultado satisfatório e com isso foi autodestruído, concluindo este primeiro teste.

Figura 33 – Fim primeiro teste

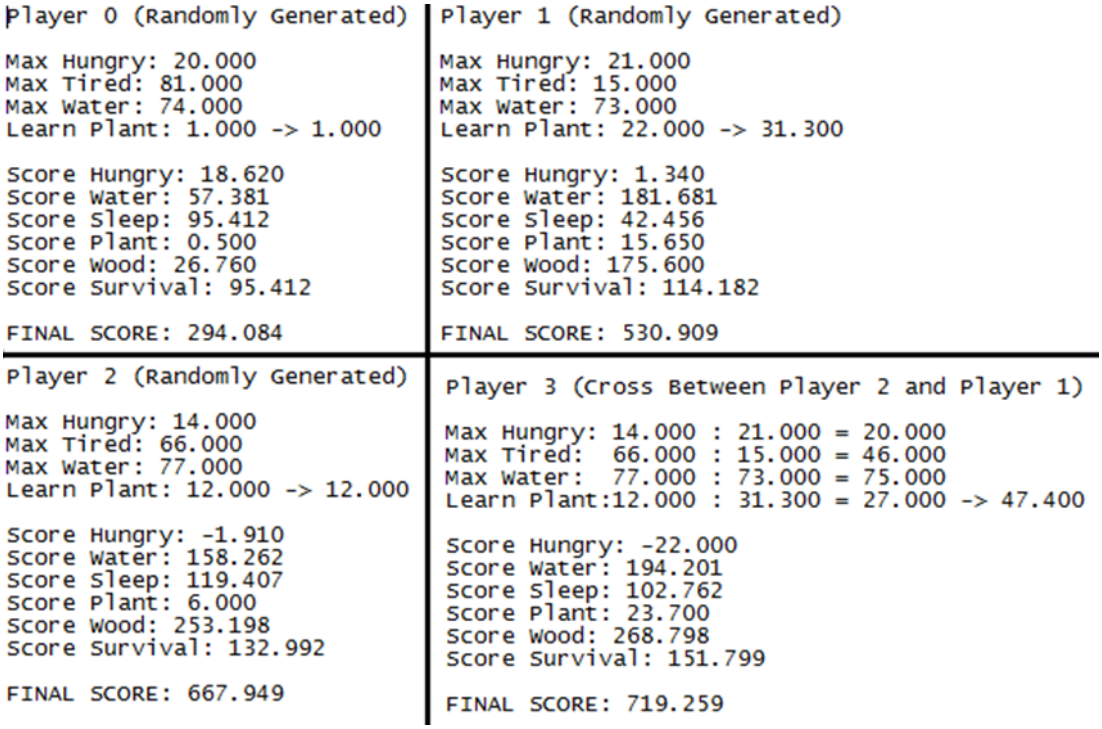


Fonte: Elaborada pelo autor

A imagem que antecedeu sua autodestruição mostra que o agente tinha uma plantação de frutas perto do lago e da sua casa.

O segundo teste foi parecido com o primeiro, por também esperar o resultado de uma mutação em um de seus atributos. Seguem os primeiros seres:

Figura 34 – Segundo teste, primeiros agentes



Fonte: Elaborada pelo autor

Já nos três primeiros *players*, nota-se que o atributo *Max Hungry* irá ficar abaixo de trinta novamente. Com isso, esperava-se a mutação com um bom valor, porém, enquanto isso não ocorria, os seres foram melhorando seu aprendizado de plantação, mas os demais atributos continuaram parecidos.

A mutação que deixou o valor maior que trinta, veio com o *player* 53, como segue ilustrado, junto com os outros seres gerados:

Figura 35 – Segundo teste, últimos agentes

```
Player 53 (Cross Between Player 41 and Player 38)

Max Hungry: 17.000 : 17.000 = (M) 33.000
Max Tired:  52.000 : 52.000 = 52.000
Max Water:  75.000 : 75.000 = 75.000
Learn Plant:129.800 : 122.800 = 127.000 -> 139.500

Score Hungry: 209.125
Score Water: 700.173
Score Sleep: 339.241
Score Plant: 3616.000
Score Wood: 468.395
Score Survival: 462.240

FINAL SCORE: 5795.173
```

```
Player 54 (Cross Between Player 53 and Player 41)

Max Hungry: 33.000 : 17.000 = 19.000
Max Tired:  52.000 : 52.000 = 52.000
Max Water:  75.000 : 75.000 = 75.000
Learn Plant:139.500 : 129.800 = 132.000 -> 132.000

Score Hungry: 0.720
Score Water: 0.480
Score Sleep: 0.240
Score Plant: 1056.000
Score Wood: 0.000
Score Survival: 0.240

FINAL SCORE: 1057.680
```

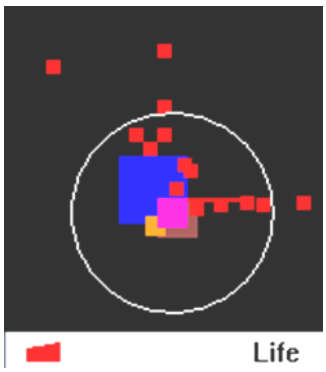
```
Player 55 (Cross Between Player 53 and Player 41)

Max Hungry: 33.000 : 17.000 = 31.000
Max Tired:  52.000 : 52.000 = 52.000
Max Water:  75.000 : 75.000 = (M) 44.000
Learn Plant:139.500 : 129.800 = 137.000 -> 143.300

Score Hungry: 113.622
Score Water: 204.301
Score Sleep: 184.597
Score Plant: 3552.800
Score Wood: 449.595
Score Survival: 251.032

FINAL SCORE: 4755.947
```

```
Player 56 (Cross Between Player 53 and Player 55)

Max Hungry: 33.000 : 31.000 = 31.000
Max Tired:  52.000 : 52.000 = 52.000
Max Water:  75.000 : 44.000 = 72.000
Learn Plant:139.500 : 143.300 = 139.000 -> 200.000

Score Hungry: 1435.079
Score Water: 2929.143
Score Sleep: 1439.977
Score Plant: 18780.000
Score Wood: 593.595
Score Survival: 1599.015

FINAL SCORE: 26776.809 (SATISFACTORY RESULT!!!)
```

Fonte: Elaborada pelo autor

O resultado satisfatório não apareceu assim que ocorreu a mutação, pelo fato de ainda não ter tanta habilidade na plantação, mas com a passagem de mais algumas gerações, o *player* 56 foi satisfatório, porém, com uma ressalva, como mostrado na figura a seguir:

Figura 36 – Fim segundo teste

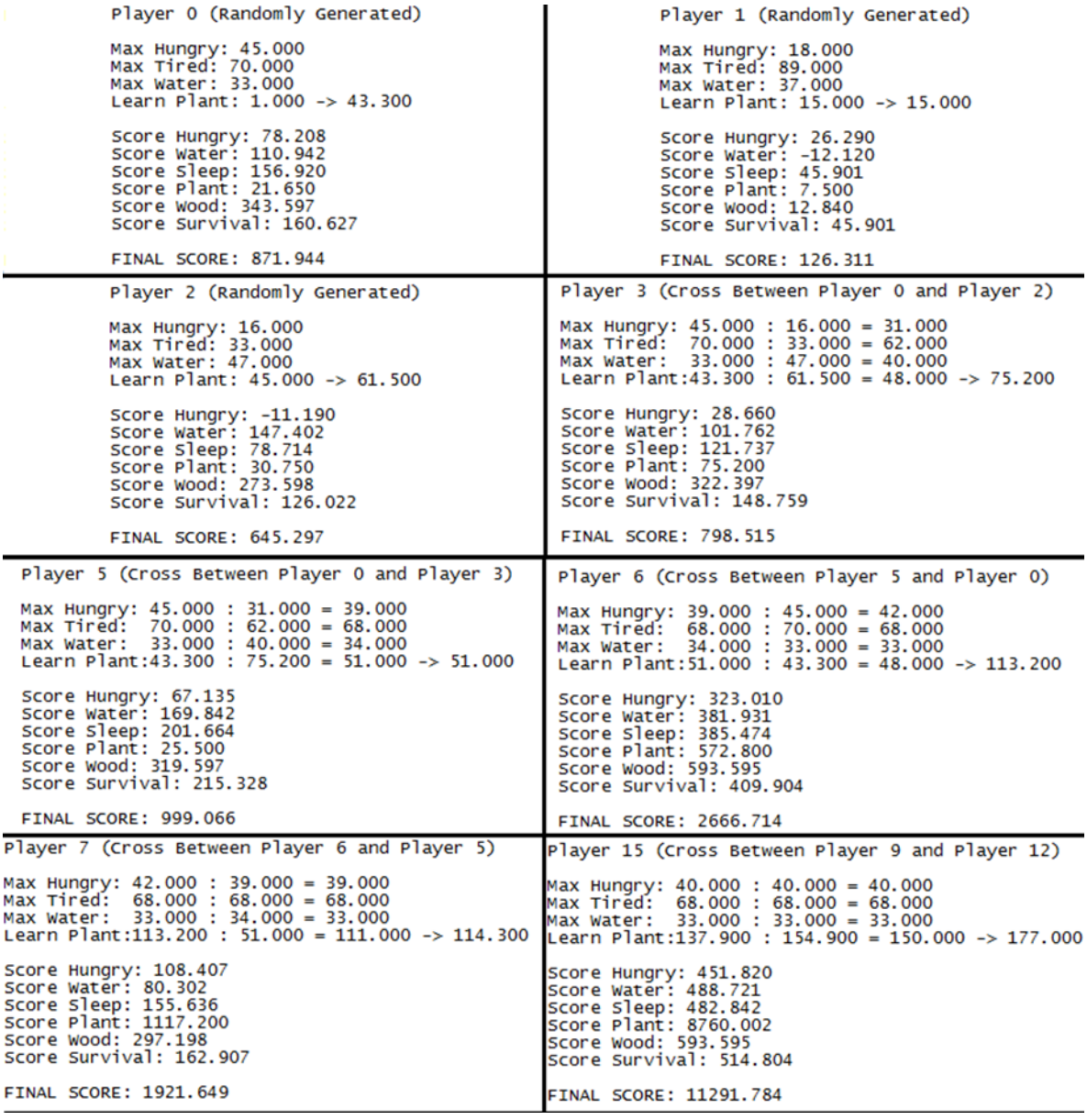


Fonte: Elaborada pelo autor

Por ter seu *Max Hungry* muito perto do limite, muitas vezes o valor do mesmo foi abaixo de trinta, fazendo com que o player perdesse vida, enquanto não se fixou perto de um lago, com sua casa construída. Mas, no momento em que isso aconteceu, ele não perdeu mais vida, pois tinha tudo o que precisava muito próximo.

No terceiro teste, os primeiros seres apresentaram bons atributos, facilitando muito para as próximas gerações, que não iriam precisar de uma mutação para chegar ao resultado satisfatório. Seguem os principais agentes deste teste, conforme mostrado a seguir:

Figura 37 – Terceiro teste, principais agentes



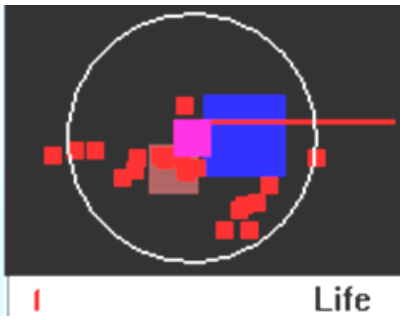
```
Player 16 (Cross Between Player 15 and Player 9)

Max Hungry: 40.000 : 40.000 = 40.000
Max Tired:  68.000 : 68.000 = 68.000
Max Water:  33.000 : 33.000 = 33.000
Learn Plant:177.000 : 137.900 = 155.000 -> 192.000

Score Hungry: 724.719
Score Water: 613.063
Score Sleep: 611.725
Score Plant: 11624.000
Score Wood: 593.595
Score Survival: 639.666

FINAL SCORE: 14806.767
```

```
Player 17 (Cross Between Player 16 and Player 15)

Max Hungry: 40.000 : 40.000 = 40.000
Max Tired:  68.000 : 68.000 = 68.000
Max Water:  33.000 : 33.000 = 33.000
Learn Plant:192.000 : 177.000 = 177.000 -> 200.000

Score Hungry: 1860.612
Score Water: 1446.347
Score Sleep: 1391.213
Score Plant: 21620.000
Score Wood: 593.595
Score Survival: 1435.486

<<Self-Destruction>>
FINAL SCORE: 28347.252 (SATISFACTORY RESULT!!!)
```

Fonte: Elaborada pelo autor

Porém, o resultado final acabou sendo parecido com o do segundo teste. Por causa do atributo *Max Water* estar perto de trinta, ele também acabou perdendo bastante vida, como mostra a imagem no final da simulação:

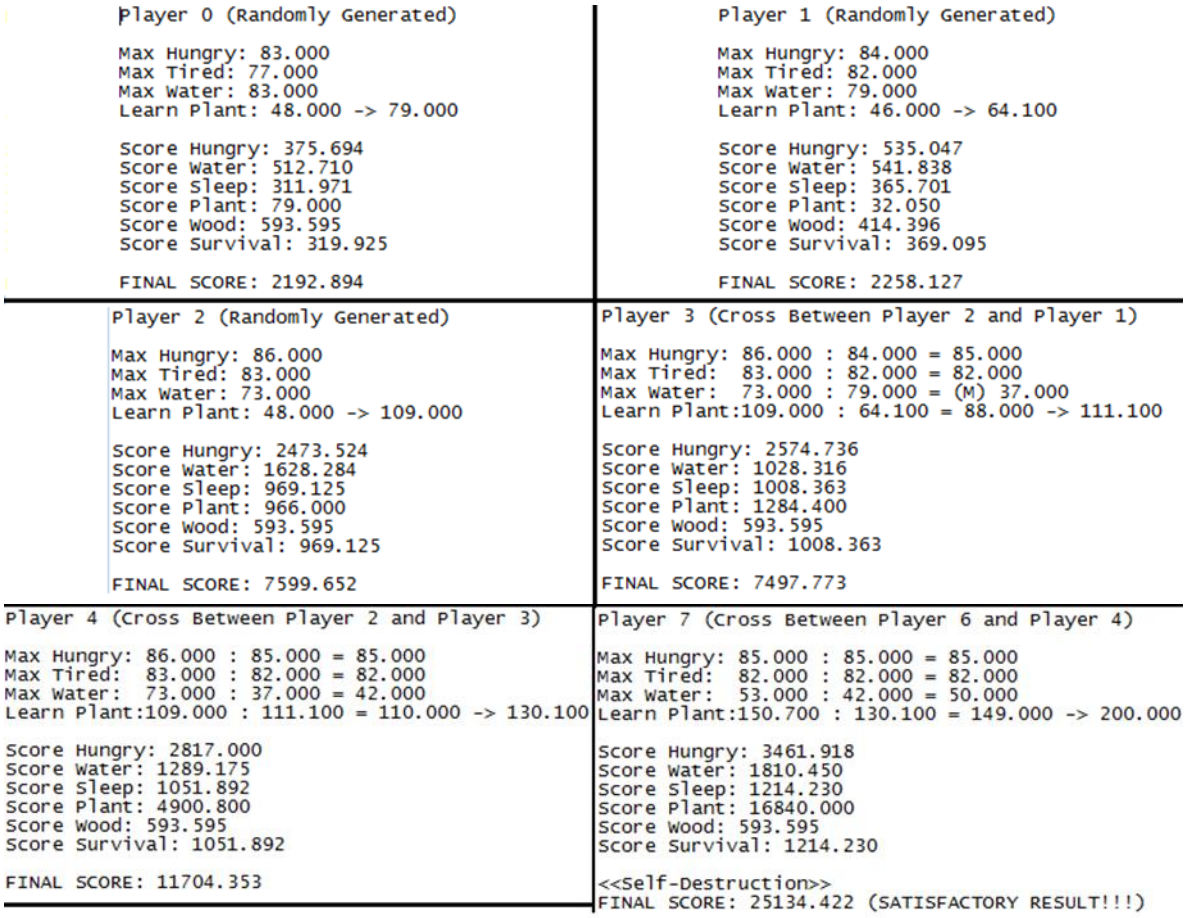
Figura 38 – Fim terceiro teste



Fonte: Elaborada pelo autor

No quarto teste, nos seres gerados aleatoriamente, os valores dos atributos foram forçados a serem altos, sempre dentro do escopo pré-definido antes, entre dez e noventa, para verificar o que pode acontecer em uma situação limite máxima. Seguem os principais seres gerados no quarto teste, conforme mostra a Figura 39:

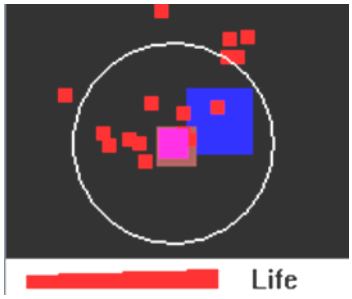
Figura 39 – Quarto teste, principais agentes



Fonte: Elaborada pelo autor

Como resultado, visto que o rendimento foi muito bom, comparando com os primeiros *players* de cada um dos outros testes. Entretanto, percebeu-se que por causa de estar constantemente tendo que cuidar de seus atributos, o agente demora um pouco mais para ter a casa finalizada, mas no final da simulação, sua vida está cheia, por ter um grande escopo entre os atributos e o valor mínimo de trinta, no caso do *player* 7, o escopo do *Max Hungry* é de cinquenta e cinco. Segue a Figura 40 para ilustrar a situação final:

Figura 40 – Fim quarto teste



Fonte: Elaborada pelo autor

Concluindo, pode-se dizer, com base nos testes realizados, que o importante para se chegar a um resultado satisfatório é ter seus atributos maiores que trinta e quanto maior, dentro do escopo estipulado, melhor é o seu resultado final. Se apenas um dos atributos for perto de trinta, mas os outros forem maiores, o player consegue sobreviver, mas até se fixar a um lugar, perde muita vida.

5 Conclusão

Com base nos testes e todo o estudo realizado sobre Algoritmos Genéticos no decorrer deste trabalho, demonstrou-se que no final, bons resultados sempre são obtidos, sem levar em consideração a quantidade de jogadores ou o tempo, pois como em alguns casos é preciso que haja uma boa mutação para se obter o resultado satisfatório, o tempo para se chegar a esse resultado pode variar bastante.

Vale salientar, também, que para ter resultados coerentes, o modo em que é implementada a pontuação do agente inteligente deve estar bem balanceada, o que só pode ocorrer com vários testes realizados no simulador, analisando resultados e percebendo alguns padrões, para saber quanto cada ação, certa ou errada, deve valer. Se não, pode ocorrer do algoritmo genético escolher um ser que foi claramente pior que outro, mas por ter a pontuação mal implementada, acabou sendo como um dos melhores seres, o que não seria correto.

Conclui-se que é interessante a implementação de algoritmos genéticos nos jogos digitiais, possibilitando a cada partida se adequar ao jogador, tornando-se uma experiência única para cada pessoa.

A cada NPC gerado, bem ou mal sucedido durante a partida, seriam guardados seus atributos e suas ações para serem analisados, verificando as melhores para serem mantidas para as próximas gerações e as piores, para que não se repitam mais ou apareçam com menos frequência, assim, moldando as personagens controladas pelo computador as necessidades ou habilidades do jogador.

A inteligência artificial em jogos ainda tem muito para ser estudada, por mais avanços que já ocorreram. Cada vez mais os jogadores buscam ter uma experiência com um NPC, sendo aliado ou inimigo, com um comportamento parecido ao de um ser humano, como se um outro jogador estivesse controlando esta personagem.

Foi nesta ideia, que o presente trabalho se apoiou, tentando fazer com que cada ser tivesse um comportamento parecido a de um humano, dentro de suas necessidades e habilidades.

Algumas sugestões de continuação deste trabalho seriam:

- Implementar Busca A*: que permitiria que o *player* desviasse dos objetos como lagos, árvores e alimentos ao invés de passar por cima deles.

- Colocar mais agentes no mesmo ambiente: programar interação entre mais agentes diferenciando-os por sexo, masculino ou feminino, verificando o que aconteceria, como por exemplo, iriam se ajudar, dividir tarefas, formar tribos, continuar a trabalhar sozinho, brigar por território, entre muitas outras possibilidades.

- Melhorar interface: fazer *sprites* para os *players* e objetos do cenário, mostrar o que o agente está fazendo na janela de execução do jogo e não mais no *debug*.

- Adicionar sons: como por exemplo o som do ambiente, das ações executadas pelo agente (andar, comer, construir casa, etc), fim do jogo com e sem resultado satisfatório.

- Refinar o AG: para que o mesmo não escolha sempre os dois melhores seres até o momento para se cruzarem, dando uma chance a cada agente já gerado de ser escolhido, referente a sua pontuação. Desse modo não dependerá apenas da mutação para que haja maior variabilidade entre as espécies da população.

Referências Bibliográficas

ANDRADE, K. O.; SILVA, A. E. A.; CROCOMO, M. K. Um Algoritmo Evolutivo para adaptação de NPCs em um jogo de ação. Escola de Engenharia de Piracicaba, 2008. Disponível em: <http://kleberandrade.files.wordpress.com/2012/04/andrade2009.pdf>. Acesso em: 10 out. 2013.

BELISARIO, L. F. B.; MEDINA, J. Algoritmos Genéticos com Métodos de Aprendizado no Desenvolvimento de Jogos. Universidade Estadual do Mato Grosso do Sul, dez. 2010.

BRAGA, B. T. R.; PERREIRA, J. L. A. Agentes Inteligentes – Conceitos, Características e Aplicações. Universidade da Amazônia. Belém – PA, 2001. Disponível em <http://www.trabalhosfeitos.com/ensaios/Agentes-Inteligentes-Conceitos-Caracter%C3%ADsticas-e/50648501.html>. Acessado em: 11 abr. 2014.

CPLUSPLUS. Thread, 2000; Disponível em <http://www.cplusplus.com/reference/thread/thread/– acesso tal dia>. Acesso em: 5 abr. 2014.

FERREIRA, A. B. H. Novo Dicionário da Língua Portuguesa, Primeira Edição. Rio de Janeiro, Nova Fornteira, 1975, 1499 p.

JOHANN, M. O. Curso de Introdução à Programação em C++, UFRGS, Porto Alegre - RS, ago. 2004. Disponível em <http://www.inf.ufrgs.br/~johann/cpp2004/>. Acesso em: 29 out. 2013.

MANSSOUR, I. H. Introdução à OpenGL, PUCRS, Porto Alegre – RS, mar. 2003. Disponível em <http://www.inf.pucrs.br/~manssour/OpenGL/Introducao.html>. Acesso em: 29 out. 2013.

OLIVEIRA, V. A. P.; SOARES, P. L. B. Brian – Um agente inteligente cognitivo e deliberativo para aplicações em jogos de computadores.Centro Universitário de Volta Redonda - RJ, 2010.

PAUL, J. M. A Origem das Espécies. Tradução do primeiro volume de Charles Darwin., Porto, Lello & Irmão, 2003, 572 p. Disponível em <http://ecologia.ib.usp.br/ffa/arquivos/abril/darwin1.pdf>. Acesso em: 12 fev. 2014.

PRIBERAM, Inteligência Artificial. Em: Priberam Dicionário Online. Disponível em <http://www.priberam.pt/dlpo/intelig%C3%AAncia>. Acesso em: 9 out. 2013.

RICARTE, I. L. M. Programação Orientada a Objetos: Uma Abordagem com Java. UNICAMP, Campinas – SP, 2001. Disponível em <http://www.dca.fee.unicamp.br/cursos/PooJava/Aulas/poojava.pdf>. Acesso em: 11 maio 2014.

RUSSELL, S.; NORVIG, P. Inteligência Artificial. Tradução da segunda edição. Rio de Janeiro, Editora Campus, 2004, 1021 p.

SLOAN, ROBERT. Computer smart as a 4-year-old, UIC, Chicago, jul.2013. Disponível em <http://news.uic.edu/a-computer-as-smart-as-a-four-year-old>. Acesso em: 9 out. 2013.

SOBRINHO, M. B. Tutorial de Utilização de OpenGL. Uni-BH, Belo Horizonte – MG, 2003. Disponível em <http://pt.scribd.com/doc/2878907/Tutorial-OpenGL>. Acesso em: 16 abr. 2014.

VALENTE, L. OpenGL – Um tutorial, Universidade Federal Fluminense – Niterói – RJ, 2004. Disponível em <http://www2.ic.uff.br/PosGraduacao/RelTecnicos/282.pdf>. Acesso em: 16 abr. 2014.